MANUEL DE REFERENCE DU LANGAGE *B*

VERSION 1.8.10

MANUEL DE REFERENCE
DU LANGAGE B

Version 1.8.10 du 18 décembre 2015

Nous remercions Jean-Raymond Abrial pour sa contribution
dans la réalisation de ce manuel.

Document établi par ClearSy

Si vous constatez des erreurs ou des imprécisions dans ce document,
merci de nous en faire part à l'adresse suivante :

e-mail : maintenance.atelierb@clearsy.com

Tél. : (+33) 04.42.37.12.97

Fax : (+33) 04.42.37.12.71

ClearSy
Support Atelier B

Parc de la Duranne - 320, avenue Archimède

Les Pléiades 3 – Bât A
13857 Aix-en-Provence Cedex 3
FRANCE

SOMMAIRE

SUIVI DES MODIFICATIONS

Manuel de Référence du Langage B version 1.8.10 (Version livrée avec l’Atelier B version 4.3.0)

1. Ajout des opérateurs real, ceiling et floor pour les nombres réels.

Manuel de Référence du Langage B version 1.8.9 (Version livrée avec l’Atelier B version 4.2.0)

2. Ajout des types réels et flottants et de de l’impact sur les expressions numériques.

Manuel de Référence du Langage B version 1.8.8 (Version commerciale livrée avec l’Atelier B version 4.0.1)

1. Correction des références aux annexes dans l'introduction.

Manuel de Référence du Langage B version 1.8.7 (Version commerciale livrée avec l’Atelier B version 4.0)

Objectif : intégration des remarques pour l'Atelier B v4.0.

1. Suppression de la définition de closure0.

Manuel de Référence du Langage B version 1.8.6 (Version commerciale livrée avec l’Atelier B version beta 3.7)

Objectif : intégration des remarques pour l'Atelier B v3.7.

1. Suppression de la référence aux bijections partielles.

Manuel de Référence du Langage B version 1.8.5 (Version commerciale livrée avec l’Atelier B version 3.6)

Objectif : intégration des remarques pour l'Atelier B v3.6.

1. Précision concernant le modèle équivalent des opérations locales.
2. Restriction concernant l’ensemble vide.
3. Modification de la syntaxe de la composition de relation et du produit parallèle conformément aux analyseurs syntaxique existants.
4. Ajout d’une restriction pour l’appel d’opération $x, x \longleftarrow \mathrm{op}$.

Manuel de Référence du Langage B version 1.8.4 (Version commerciale livrée avec l’Atelier B version beta 3.6)

Objectif : intégration des remarques des utilisateurs depuis la dernière version commerciale 1.8.1.

1. Contrainte sur les entiers littéraux : pointeur dans la partie lexicale sur la partie syntaxique.

2. Correction dans la définition de l'opérateur mod.
3. Correction de l'exemple sur l'opérateur puissance.
4. Correction du typage de l'expression rel (R).
5. Correction de la définition d'une suite en extension.
6. Correction de la bonne définition des opérateurs : first, last, front, tail, ↑ et ↓ et modification de la description de ↑ et ↓.
7. Correction de la définition de rev (S).
8. Correction de la définition de ←.
9. Modification de la description de l'instruction « devient égal ».
10. Correction dans la syntaxe du CASE et changement dans l'ordre des productions sur les substitutions.
11. Correction de la bonne définition de const (x, q) et de infix (t).
12. Modification de la définition de cat.
13. Correction de l'exemple utilisant bin.
14. La clause SETS est interdite au sein d'une définition.
15. Suppression de la restriction sur le typage des paramètres ensembles d'une machine. Il suffit d'indiquer que ce sont des types et d'appliquer les règles de vérification de typage.
16. Mise à jour des priorités des opérateurs B données en annexe, selon les priorités définies dans l'Atelier B.
17. Correction de la définition de conc (S).
18. Ajout du type de sizet (t).
19. Ajout de productions grammaticales sur les expressions d'arbres.
20. Correction sur les restrictions d'arbres binaires.
21. Correction sur la définition des ∑ ou des ∏ vides.
22. Précision dans les tables de visibilité sur la nature non homonyme des données propres à un composant.
23. Correction sur la description de la division entière.
24. Assouplissement de la restriction sur l'appel en parallèle de deux opérations d'une machine *incluse* : seul l'appel d'opérations de modification est interdit.
25. Précision sur les arbres : un arbre n'est jamais vide.
26. Ajout de λ dans la liste des expressions introduisant des variables quantifiées.
27. Changement de syntaxe pour les expressions utilisant plusieurs paramètres. Avant on utilisait une seule expression, sachant que ',' est elle-même une expression de couple, maintenant on utilise N expressions séparées par des ','.

28. Rajout dans terme simple de bool et d'accès à un élément de tableau ou de record dans les expressions arithmétiques.

29. Modification de la table de visibilité des opérations locales dans une implantation par rapport à son abstraction.

30. Regroupement dans l'index des opérateurs B sous la rubrique '#'.

31. La portée du changement de type des ensembles abstraits, lors de leur valuation en implantation, s'étend désormais à la clause PROPERTIES.

Manuel de Référence du Langage B version 1.8.3 (Atelier B version 3.6 interne)

Objectif : intégration des opérations locales pour l'Atelier B v3.5.

1. Typage : tableau de typage des données, typage des paramètres d'entrée d'opération et typage des paramètres de sortie d'opération.

2. Substitution simultanée : modification de la contrainte sur la modification en parallèle de variables distinctes, insertion de la restriction sur les appels en parallèle d'opérations *importées*.

3. Clause OPERATIONS : nouvelles restrictions, la clause contient désormais également les implémentations des opérations locales.

4. Clause LOCAL_OPERATIONS : nouvelle clause pour spécifier les opérations locales.

5. Contrôles d'anticollision d'identificateurs.

6. Restrictions de l'Atelier B : nouvelle restriction sur les paramètres de sortie en rapport avec les opérations locales (FT2229).

7. Glossaire : modification des définitions d'opération et d'opération propre, ajout d'opération locale.

8. Tables de visibilité d'implantation : une nouvelle colonne pour la clause LOCAL_OPERATIONS et une nouvelle ligne pour les opérations locales.

9. Nouveau mot réservé LOCAL_OPERATIONS, en annexe et dans la feuille des symboles B.

Manuel de Référence du Langage B version 1.8.2 (Atelier B version 3.5)

Objectif : corrections des erreurs et des imprécisions détectées principalement par l'équipe B. Version de référence pour l'Atelier B v3.5.

1. précision concernant le $0 dans une boucle WHILE : s'applique à une variable implantée par homonymie et modifiée dans le corps de la boucle (5.1 Expressions primaires, restriction 3, § sur la boucle tant que et 6.17 Substitution boucle tant que, description, § sur le $0).

Manuel de Référence du Langage B version 1.8.1 (Version commerciale livrée avec l'Atelier B version 3.5)

Objectif : corrections des erreurs détectées par les beta testeurs de l'Atelier B v1.5 et intégrations de certaines remarques faites par le comité de relecture du manuel lors de la réunion n°6, du 30/03/98.

1. corrections concernant l'intégration des arbres dans la grammaire B (index et BNF),

2. correction concernant les définitions : ~~partie droite~~ → le corps,
3. corrections concernant la BNF (cf. bordereau de relecture v1.8),
4. ~~ensemble abstrait~~ → ensemble différé,
5. ~~termes~~, ~~condition~~, ~~instruction~~ → expression B0, prédicat B0, substitution B0,
6. table de visibilité : ~~lecture écriture~~ → visible,
7. correction des définitions de chaîne littérale et commentaire,
8. ~~plongement~~ → changement de type,
9. correction, $0 dans les ASSERT de boucles,
10. modification des définitions des substitutions appel d'op. et boucle : ~~[S1][S2]~~ → [S1 ; S2],
11. suppression des contrôles syntaxiques de présence des clauses PROPERTIES et INVARIANT,
12. modification de visibilité des valuations,
13. ~~□truct~~ → struct,
14. correction de la grammaire ~~f ; g~~ → (f ; g) et ~~f || g~~ → (f || g),
15. ajout des expressions rec comme mélange avec et sans labels,
16. correction de la définition de ⤗,
17. modification des règles de syntaxe de la forme ~~(exp, exp)~~ → (exp),
18. ajout de la règle (expression_arithmétique),
19. correction nom des substitutions dans la grammaire,
20. suppression des unions de tableaux concrets.

Manuel de Référence du Langage B version 1.8 (Atelier B version 3.5.beta)

Objectif : évolutions du langage supportées par l'Atelier B v3.5 : fichiers de définitions, records et non supportées par l'Atelier B v3.5 : les arbres.

1 INTRODUCTION

Avant-propos

Le *Manuel de Référence du Langage B* décrit le langage B supporté par l'outil **Atelier B** version 4.3.0. Ce langage est fondé sur le langage présenté dans l'ouvrage de référence *The B-Book*, cependant certaines avancées comme la récursivité ou le raffinement multiple ne sont actuellement pas prises en compte par le langage B.

Historique

La méthode B est une méthode formelle permettant le développement de logiciels sûrs. Elle a été conçue par Jean-Raymond ABRIAL, qui avait déjà participé dans les années 1980 à la conception de la notation Z. Les contributeurs de la Méthode B sont trop nombreux pour être remerciés ici, on en trouvera une liste dans les premières pages du *B-Book*. D'autre part, la méthode B repose sur les travaux scientifiques menés à l'université d'Oxford, dans le cadre du *Programming Research Group* dirigé par C.A.R. Hoare. Le *B-Book* de J.R. Abrial est l'ouvrage fondamental décrivant la méthode B.

Objectif

L'objectif de ce document est de définir précisément le langage B afin d'en constituer le Manuel de Référence. Il est principalement destiné aux utilisateurs qui réalisent des développements selon la méthode B, mais aussi à tous ceux qui souhaitent découvrir les possibilités du langage B. Le langage décrit dans ce document ne constitue pas en lui-même une norme, mais il tend à s'en rapprocher le plus possible.

Présentation de la méthode B

Le développement d'un projet selon la méthode B comporte deux activités étroitement liées : l'*écriture* de textes formels et la *preuve* de ces mêmes textes.

L'activité d'écriture consiste à rédiger les spécifications formelles de *machines abstraites* à l'aide d'un formalisme mathématique de haut niveau. Ainsi, une spécification B comporte des données (qui peuvent être exprimées entre autres par des entiers, des booléens, des ensembles, des relations, des fonctions ou des suites), des *propriétés invariantes* portant sur ces données (exprimées à l'aide de la logique des prédicats du premier ordre), et enfin des *services* permettant d'initialiser puis de faire évoluer ces données (les transformations de ces données sont exprimées à l'aide de substitutions). L'activité de preuve d'une spécification B consiste alors à réaliser un certain nombre de démonstrations afin de prouver l'établissement et la conservation des propriétés invariantes en question (par exemple il faut prouver que l'appel d'un service conserve bien les propriétés invariantes). La génération des assertions à démontrer est complètement systématique. Elle s'appuie notamment sur la transformation de prédicats par des substitutions.

Le développement d'une machine abstraite se poursuit par une extension de l'activité d'écriture lors d'étapes successives de *raffinement*. Raffiner une spécification consiste à la reformuler en une expression de plus en plus concrète, mais aussi à l'enrichir. L'activité de preuve concernant les raffinements consiste également à réaliser un certain nombre de vérifications statiques et à prouver que le raffinement constitue bien une reformulation valide de la spécification. Le dernier niveau de raffinement d'une machine abstraite se nomme l'*implantation*. Il est assujetti à quelques contraintes supplémentaires : par exemple il ne peut plus manipuler que des données ou des substitutions ayant un équivalent informatique. Les données et les substitutions de l'implantation constituent un langage informatique similaire à un langage impératif. À ce titre, il peut donc s'exécuter sur un système informatique après fabrication d'un exécutable, soit à l'aide d'un compilateur dédié soit en passant par une étape intermédiaire de traduction automatique vers Ada, Ada sécuritaire, C++ ou C.

Guide de lecture

Nous procédons maintenant à un rapide tour d'horizon de notre document.

Le chapitre 2, *Concepts de base*, présente les principes de l'analyse formelle d'un texte rédigé en langage B : analyse lexicale, analyse syntaxique et analyse sémantique. Il donne les conventions lexicales et décrit les différentes sortes d'unités lexicales. Enfin, il présente les conventions syntaxiques utilisées dans le reste du document afin de décrire la grammaire du langage B.

Le chapitre 3, *Typage*, présente les différentes formes de données que l'on peut décrire en B. Après avoir introduit les types de B, il décrit la façon dont s'exprime le typage des données au moyen de prédicats de typage. Enfin, il présente le cas particulier du contrôle de type des tableaux.

Le chapitre 4, *Prédicats*, présente le langage des prédicats.

Le chapitre 5, *Expressions*, présente le langage des expressions.

Le chapitre 6, *Substitutions*, présente le langage des substitutions.

Le chapitre 7, *Composants*, décrit clause par clause le corps des composants B, c'est-à-dire, les machines abstraites, les raffinements et les implantations. Il présente également les règles d'anticollision des identificateurs au sein des composants.

Le chapitre 8, *Architecture B*, présente l'architecture générale d'un projet B. Il décrit les modules, leurs composants (les machines abstraites, les raffinements et les implantations), les liens qui existent entre les composants. Enfin, il présente les bibliothèques.

L'annexe A, *Mots réservés et opérateurs*, présente la table des mots clés et la table des opérateurs avec leurs priorités.

L'annexe B récapitule l'ensemble de la grammaire du langage B.

L'annexe C, *Tables de visibilité*, regroupe les règles de visibilité des constituants d'un composant par rapport aux composants auxquels il est relié.

L'annexe D constitue le *Glossaire*.

L'annexe E constitue l'*Index*.

2 CONCEPTS DE BASE

Ce chapitre présente les principes généraux de l'analyse formelle du langage B, ainsi que les conventions lexicales et syntaxiques adoptées dans le reste du document.

Un projet B est constitué d'un certain nombre de composants (cf. chapitre 7 *Composants*). Chaque composant est stocké dans un fichier séparé. L'analyse d'un composant se scinde en trois parties successives : l'analyse lexicale, l'analyse syntaxique et l'analyse sémantique.

Analyse lexicale

L'analyse lexicale consiste à vérifier que le composant est constitué d'une suite de *lexèmes* valides et à effectuer l'analyse et le remplacement des définitions textuelles (cf. §2.3 *La clause DEFINITIONS*). À cette occasion sont définis les éléments du vocabulaire terminal du Langage B, comme par exemple les identificateurs.

Analyse syntaxique

L'analyse syntaxique permet de vérifier que la suite de lexèmes qui constitue un composant respecte les règles de production de la grammaire du langage B. Ces règles sont rassemblées dans l'annexe B.1 *Grammaire du langage B*.

Analyse sémantique

Enfin, l'analyse sémantique permet de vérifier que le composant possède un sens conforme à la Méthode B. En B, l'analyse sémantique se décompose en deux phases, une phase de vérification statique et une phase de preuve. La phase de vérification statique réalise les contrôles automatiques décrits ci-dessous :

- **le contrôle de typage des expressions** permet de vérifier que les données sont correctement typées et que les expressions utilisées au sein de prédicats, d'expressions ou de substitutions possèdent des types compatibles (cf. §3.1 *Fondements du typage*). Ces contrôles sont spécifiés dans la rubrique « Règles de typage » de chaque prédicat, expression ou substitution,
- **la résolution de portée** permet de déterminer, lors de l'utilisation d'une donnée, à quelle déclaration elle se rattache. La résolution de portée s'effectue à l'aide des « Règles de portée » spécifiées dans une rubrique spéciale pour chaque prédicat, expression ou substitution,
- **le contrôle de visibilité** permet de vérifier que les données d'un composant sont utilisées au sein des clauses de ce composant selon un mode d'accès correct (accès en lecture ou en écriture). Les modes d'accès autorisés sont spécifiés dans les tables de visibilité (cf. Annexe C),
- **les contrôles d'anticollision d'identificateurs** permettent d'éviter toute ambiguïté lors de l'utilisation d'une donnée (cf. §7.26 *Règles d'anticollision d'identificateurs*),
- **les restrictions sémantiques** décrites dans les rubriques « Restrictions » des chapitres *Prédicats*, *Expressions*, *Substitutions* et *Composants* donnent la liste des contrôles statiques qui ne sont pas pris en compte par les contrôles décrits ci-dessus.

La phase de preuve permet de démontrer certaines propriétés pour lesquelles on n'a pas de procédure de décision. Ces propriétés sont appelées *Obligations de Preuves*. Elles sont générées de manière systématique à partir de composants B déjà vérifiés statiquement. Pour qu'un composant B soit déclaré correct, il faut que toutes ses

Obligations de Preuves aient été prouvées par une démonstration mathématique.

Dans ce document, les vérifications sémantiques statiques sont décrites précisément. Quant aux Obligations de Preuve, elles sont évoquées mais ne sont pas détaillées (cf. *Manuel de Référence des Obligations de Preuve*).

2.1 Conventions lexicales

En B, il existe quatre sortes d'unités lexicales :

- les mots réservés et les opérateurs,
- les identificateurs,
- les entiers littéraux et réels,
- les chaînes de caractères littérales.

L'analyse lexicale d'un composant consiste à décomposer son texte en une suite de lexèmes du début du texte jusqu'à la fin, tout en éliminant les caractères d'espacement inutiles et les commentaires.

Le formalisme retenu pour décrire les unités lexicales est celui de l'analyseur lexical *LEX*, dont nous rappelons les conventions :

Expression régulière	**Chaîne de caractères**	**Exemple**	
x	le caractère *x*	b	: la lettre b
[*x*]	le caractère *x*	[-]	: le signe moins
[xy]	le caractère *x* ou le caractère *y*	[bB]	: lettre b ou la lettre B
[*x*-*y*]	un caractère de l'intervalle *x*..*y*, selon l'ordre ASCII	[a-z]	: une lettre minuscule
[^*x*]	tout caractère sauf *x*	[^\"]	: pas de guillemet
x?	*x* facultatif	[\-]? 0	: 0 ou -0
x^*	le caractère *x* répété de 0 à *n* fois	$[0\text{-}9]^*$	: un entier positif ou rien
x^+	le caractère *x* répété de 1 à *n* fois	$[0\text{-}9]^+$	: un entier positif
.	n'importe quel caractère sauf le caractère saut de ligne		

Voici la description des unités lexicales ainsi que des caractères d'espacement et des commentaires.

Les mots réservés et les opérateurs

Les mots réservés et les opérateurs sont formés d'une suite non vide de caractères imprimables. Leur liste est donnée en Annexe A. Les symboles mathématiques employés dans le Langage B possèdent tous un équivalent en caractères ASCII. Pour faciliter la lecture de ce document, seuls les symboles mathématiques seront utilisés. La correspondance entre les deux notations est donnée en Annexe A.

Afin de simplifier la syntaxe du langage, on associe à tous les opérateurs un ordre de priorité ainsi qu'une associativité (à gauche ou à droite). Ces deux propriétés permettent de lever toute ambiguïté lors de l'analyse syntaxique d'une expression ou d'un prédicat composé de plusieurs opérateurs.

Les identificateurs

Ident : $[a\text{-}zA\text{-}Z][a\text{-}zA\text{-}Z0\text{-}9\_]^*$

Un identificateur est une séquence de lettres, de chiffres ou du caractère souligné "_". Le premier caractère doit être une lettre. Les lettres minuscules et majuscules sont distinguées. Un identificateur peut être de taille quelconque.

Le caractère point "." n'est pas autorisé pour les identificateurs. En B, le point sépare les différents préfixes de renommage d'un constituant renommé (cf. §8.3 *Instanciation et renommage*).

Les entiers littéraux

Entier_littéral : [\-]? [0-9]$^+$

Les entiers littéraux sont des suites de chiffres, éventuellement précédés du signe moins "-" pour désigner les entiers négatifs. Les entiers littéraux doivent être compris entre MININT et MAXINT (cf. §5.3 *Expressions arithmétiques*).

Les réels littéraux

Réel_littéral : [\-]? [0-9]$^+$ [.][0-9]$^+$

Les réels littéraux sont des suites de chiffres avec une partie décimale obligatoire séparée de la partie entière par un point. Ils peuvent être précédé du signe moins.

Les chaînes de caractères littérales

Chaîne_de_caractères : ["][.]*["]

Les chaînes de caractères littérales sont des suites de caractères compris entre deux caractères guillemets '"'. Tous les caractères ASCII imprimables sont acceptés à l'exception du caractère guillemet '"' qui délimite les chaînes de caractères littérales et du caractère saut de ligne.

Les commentaires

Les commentaires sont délimités par les deux caractères de début de commentaire "/*" et par les deux caractères de fin de commentaire "*/". L'intérieur du commentaire doit être constituée d'une suite de 0 à N caractères ASCII imprimables quelconques à l'exception des deux caractères consécutifs de fin de commentaire "*/". Ainsi, les commentaires ne doivent pas s'imbriquer.

Les caractères d'espacement

Les caractères d'espacement sont le caractère espace ' ', les caractères de tabulations horizontales et verticales (HT et VT), les caractères de saut de ligne (CR et LF). Les caractères d'espacement servent à séparer les lexèmes. Lorsque plusieurs caractères d'espacement se suivent, ils sont considérés comme un seul espace. Les caractères d'espacement sont obligatoires pour séparer un mot réservé d'un identificateur. Ils permettent de laisser à l'utilisateur une entière liberté quant au choix de la mise en page du texte source B.

2.2 Conventions syntaxiques

Le formalisme retenu pour la représentation de la syntaxe du langage B est une variante des formalismes BNF et EBNF dont voici les conventions :

- les mots réservés et les opérateurs sont représentés entre guillemets.
- les autres éléments du vocabulaire terminal (les identificateurs, les entiers littéraux et les chaînes de caractères littérales) sont représentés à l'aide d'une police de caractères de style normal (ni en italique, ni en gras),
- les éléments du vocabulaire non terminal sont représentés à l'aide d'une police de caractères italique,
- a ::= b désigne une production de la grammaire. a est un élément de vocabulaire non terminal et b est une suite d'éléments de vocabulaire concaténés,
- (a) désigne l'élément a,
- a | b désigne l'élément a ou l'élément b,
- [a] désigne un élément optionnel a,
- a^* désigne l'élément a concaténé n fois, où $n \geq 0$,
- a^+ désigne l'élément a concaténé n fois, où $n \geq 1$,
- a^{*b} désigne l'élément a concaténé n fois, où $n \geq 0$, avec pour séparateur l'élément b,
- a^{+b} désigne l'élément a concaténé n fois, où $n \geq 1$, avec pour séparateur l'élément b.

Attention

Les caractères " ( ) [ ] | + * font partie du métalangage de description de grammaires. Ils ne doivent pas être confondus avec les opérateurs du langage B. Ces derniers sont représentés entre guillemets, comme les autres mots réservés et opérateurs du langage B.

Exemple

Clause_abstract_variables ::= "ABSTRACT_VARIABLES" $Ident\_ren^{+","}$

Cette production de la grammaire permet de décrire le texte suivant :

ABSTRACT_VARIABLES *Var1*, *instB1.instC2.Var2*, *instD3.abstr_var*

2.3 La clause DEFINITIONS

Syntaxe

La description syntaxique de la clause DEFINITIONS utilise la notation BNF décrite au §2.2 *Conventions syntaxiques*, et non pas la notation de *LEX* donnée précédemment.

Clause_definitions ::= "DEFINITIONS" *Définition*$^{+";"}$

Définition ::= Ident ["(" Ident$^{+","}$ ")"] "==" Lexème*
| "<" Nom_de_fichier ">"
| " " " Nom_de_fichier " " "

Appel_définition ::= Ident ["(" (Lexème^{+})$^{+","}$ ")"]

Le terminal Lexème désigne n'importe quel lexème parmi les unités lexicales suivantes : les mots réservés et les opérateurs, les identificateurs, les entiers littéraux et les chaînes de caractères littérales (cf. §2.1 *Conventions lexicales*).

Le terminal Nom_de_fichier désigne un nom de fichier comprenant éventuellement un chemin relatif ou absolu, qui respecte les règles du système d'exploitation sur lequel est utilisé l'Atelier B.

Description

La clause DEFINITIONS contient des fichiers de définitions à inclure et des déclarations explicites de définitions textuelles d'un composant. Les définitions explicites peuvent être éventuellement paramétrées. Les appels de définitions situés dans le texte du composant sont remplacés lors de la phase d'analyse lexicale, avant l'analyse syntaxique. C'est pourquoi nous présentons la clause DEFINITIONS dans ce chapitre. La portée d'une définition située dans un composant est l'ensemble du composant, y compris le texte situé avant la déclaration de la définition.

Restrictions

1. Les différentes définitions d'un composant doivent avoir des noms deux à deux distincts.
2. Une définition ne doit pas utiliser les mots réservés d'en-tête de composant ni des noms de clauses. Il s'agit des mots réservés suivants : MACHINE, REFINEMENT, IMPLEMENTATION, REFINES, DEFINITIONS, IMPORTS, SEES, INCLUDES, USES, EXTENDS, PROMOTES, SETS, ABSTRACT_CONSTANTS, CONCRETE_CONSTANTS, CONSTANTS, VALUES, ABSTRACT_VARIABLES, VARIABLES, CONCRETE_VARIABLES, INVARIANT, ASSERTIONS, INITIALISATION, OPERATIONS.
3. Les éventuels paramètres formels d'une définition doivent être deux à deux distincts.
4. L'opérateur "==" est interdit dans le corps en partie droite d'une définition, c'est-à-dire la partie située après l'opérateur "==".
5. Les définitions peuvent dépendre d'autres définitions, mais elles ne doivent pas conduire à des dépendances cycliques.
6. Lors d'un appel de définition, c'est-à-dire lorsqu'un identificateur porte le nom d'une définition en dehors d'une partie gauche de définition, le nom de la définition doit être suivi d'autant de paramètres effectifs que la définition possède de paramètres formels.

7. Dans le cas de l'inclusion d'un fichier de définitions entre guillemets, le nom utilisé doit désigner un fichier à partir du répertoire courant, contenant le fichier source B.
8. Dans le cas de l'inclusion d'un fichier de définitions entre chevrons, le nom utilisé doit désigner un fichier situé à partir de l'un des répertoires de fichiers inclus.
9. Un fichier de définitions doit seulement contenir, hors commentaires, une clause DEFINITIONS respectant les règles décrites dans ce paragraphe.
10. Un fichier de définitions peut inclure d'autres fichiers de définitions, mais ces inclusions ne doivent pas conduire à des cycles.

Utilisation

Une définition est soit une référence à un fichier de définitions, soit une définition textuelle explicite. Les définitions contenues dans les fichiers de définitions sont incluses dans la liste des définitions du composant comme s'il s'agissait de définitions explicites. Ceci permet de partager des définitions entre plusieurs composants. Il suffit en effet à ces composants d'inclure le même fichier de définitions.

Si le nom d'un fichier de définitions est entouré de guillemets, le fichier est cherché à partir du répertoire local, où se situe le composant analysé. Si le nom d'un fichier de définitions est entouré de chevrons, le fichier est cherché dans l'ordre à partir de chaque répertoire de définitions. Cette liste ordonnée de répertoires est fournie par l'utilisateur à l'Atelier B.

Le nom d'une définition textuelle est un identificateur. Une définition est paramétrée si son nom est suivi par une liste d'identificateurs. Ce sont ses paramètres formels. La partie d'une définition située après l'opérateur "==" constitue le texte de remplacement de la définition. On l'appelle le corps de la définition.

Le corps d'une définition se termine lorsque l'on rencontre l'un des éléments suivants : le nom d'une clause (dont la liste est donnée dans la restriction 2), la fin d'un composant, c'est-à-dire le mot réservé END ou un caractère ';' suivi d'une autre définition.

Exemples

```
...
DEFINITIONS
  Composition (f, g)   ==   f ; g   ;
  AffectSeq (x, v)   ==   x  :=  2 × v + 1 ;
CONCRETE_CONSTANTS
...
```

Le corps de la définition *Composition* est "$f ; g$". Le dernier ";" sépare cette définition de la suivante. Le corps de la définition *AffectSeq* est "$x := 2 \times v + 1$;". Le dernier ";" fait partie de *AffectSeq* puisque la clause DEFINITIONS s'achève lors de la rencontre du mot réservé CONCRETE_CONSTANTS.

```
MACHINE
    MA
DEFINITIONS
    "commun1.def"  ;
    <commun2.def>  ;
    debut  ==  -2 × UNIT  ;
    fin  ==  10 × UNIT
SEES
...
END
```

commun1.def

```
DEFINITIONS
    UNIT  ==  16
```

commun2.def

```
DEFINITIONS
    T  ==  TRUE  ;
    F  ==  FALSE
```

La liste des définitions du composant *MA* comprend les définitions explicites *debut* et *fin* ainsi que les définitions des fichiers *commun1.def* et *commun2.def*. Le fichier *commun1.def* est cherché à partir du répertoire local, alors que *commun2.def* est cherché à partir des répertoires d'inclusion de fichiers de définitions.

Appel d'une définition

L'appel d'une définition consiste à utiliser le nom d'une définition et à fournir le même nombre de paramètres effectifs que la définition compte de paramètres formels.

Les règles de syntaxe des composants s'appliquent après l'expansion des appels de définitions textuelles.

Visibilité

Les définitions d'un composant sont locales à ce composant. Elles ne sont donc pas accessibles par les composants qui dépendent de celui-ci. Pour partager des définitions, on peut les déclarer dans un fichier de définitions et inclure plusieurs fois ce fichier.

Exemples

```
...
DEFINITIONS
    VAL_INIT  ==  -1  ;
    Somme (x1, x2)  ==  ((x1) + (x2))  ;
    BLOC_INIT  ==
        BEGIN
            Var1  :=  VAL_INIT  ;
            Var2  :=  Var1 + 1
        END ;
CONCRETE_CONSTANTS
…
```

Les parenthèses autour des paramètres formels *x1* et *x2* dans la définition de *Somme* assurent que l'on calculera effectivement la somme de *x1* et *x2* même si cette définition est appelée au sein d'une expression avec des opérateurs plus prioritaires que l'opérateur '+'.

2.4 Règles de syntaxe utiles

Les règles de syntaxe suivantes sont utilisées dans le reste du document pour simplifier la syntaxe du langage B.

Syntaxe

Liste_ident ::= Ident
| "(" Ident$^{+","}$ ")"

Ident_ren ::= Ident$^{+"."}$

Restrictions

1. Dans le cas d'une liste d'identificateurs à plusieurs éléments, les identificateurs doivent être deux à deux distincts.
2. Lorsqu'un identificateur renommé est constitué de plusieurs identificateurs, ceux-ci doivent être uniquement séparés par des caractères '.', les espaces et les commentaires sont interdits.

Description

Le non terminal *Liste_ident* représente une liste de 1 à *n* identificateurs. Si la liste comporte plusieurs identificateurs, alors elle doit être parenthésée. Une telle liste est utilisée pour déclarer des données au sein de prédicats ∀ ou ∃ ou d'expressions λ, Σ, Π, ⋃, ⋂ ou { | }.

Le non terminal *Ident_ren* représente un identificateur éventuellement renommé. Un identificateur renommé possède un préfixe constitué de 1 à *n* identificateurs séparés par le caractère point. Les identificateurs renommés désignent des données provenant de machines renommées (cf. §5.1 *Expressions primaires*).

Exemples

(*x*, *y*, *z*) est la liste des trois données : *x*, *y* et *z*.

new.var désigne la donnée *var* provenant d'une machine renommée avec le préfixe *new*.

3 TYPAGE

3.1 Fondements du typage

Le typage en B est un mécanisme de vérification statique des données et des expressions du langage B. La vérification de type d'un composant B est un pré-requis à la preuve du composant.

La notion de type B repose sur la notion d'ensemble et la propriété de monotonie de l'inclusion. Soient E une expression, S et T des ensembles tels que $S \subseteq T$. Si $E \in S$ alors $E \in T$. Le plus grand ensemble dans lequel E est contenu s'appelle le type de E.

Dans le langage B, le typage se présente sous trois aspects : les types du langage B, le typage des données et la vérification de type.

Les types du langage B

Les types possibles dans le langage B sont les types de base et les types construits à l'aide du produit cartésien, de l'ensemble des parties et des ensembles de records. Ce mécanisme est détaillé au §3.2.

Le typage des données

Dans le langage B, toute donnée qui est utilisée dans un prédicat ou une substitution doit être typée. Ce typage est réalisé lors de la première utilisation de la donnée, en parcourant le texte du prédicat ou de la substitution depuis son début. Le typage s'effectue au moyen de prédicats ou de substitutions particulières appelés prédicats de typage et substitutions de typage. On utilise un mécanisme d'inférence de type. Le type de la donnée est déduit du type des autres données intervenant dans le prédicat ou la substitution selon des règles particulières, liées au prédicat ou à la substitution. Les paragraphes suivants présentent les prédicats de typage, qui dépendent de la nature de la donnée, et les substitutions de typage.

Le tableau ci-dessous présente pour chaque nature de donnée du langage B la clause dans laquelle elle est typée et la manière de la typer.

Nature des données	**Clause de typage**	**Manière de typer**
Paramètre de machine (scalaire : identificateur avec minuscules)	clause CONSTRAINTS	prédicat de typage
Paramètre de machine (ensemble : identificateur sans minuscule)		constitue un type de base
Ensemble abstrait ou énuméré	clause SETS	constitue un type de base
Élément d'ensemble énuméré	clause SETS	typés implicitement par l'ensemble énuméré
Constante concrète	clause PROPERTIES	prédicat de typage de constante concrète
Constante abstraite	clause PROPERTIES	prédicat de typage de donnée abstraite
Variable concrète	clause INVARIANT	prédicat de typage de variable concrète
Variable abstraite	clause INVARIANT	prédicat de typage de donnée abstraite

Nature des données	**Clause de typage**	**Manière de typer**
Paramètre d'entrée d'opération	clause OPERATIONS de machine abstraite, dans une précondition	prédicat de typage de paramètre d'entrée d'opération
Paramètre de sortie d'opération	clause OPERATIONS de machine abstraite	substitution de typage
Paramètre d'entrée d'opération locale	clause LOCAL_OPERATIONS de machine abstraite, dans une précondition	prédicat de typage de paramètre d'entrée d'opération
Paramètre de sortie d'opération locale	clause LOCAL_OPERATIONS de machine abstraite	substitution de typage
Variable de prédicat ∀ ou ∃, d'expression Σ, Π, ⋃, ⋂, { | } ou λ ou de substitution ANY ou LET	toute clause qui utilise un prédicat, une expression ou une substitution	prédicat de typage de donnée abstraite
Variable locale (substitution VAR)	clause INITIALISATION ou OPERATIONS	substitution de typage

La vérification de type

Enfin, lors de l'utilisation de données déjà typées au sein d'expressions, de prédicats ou de substitutions, les règles de typage relatives à ces expressions, prédicats ou substitutions doivent être vérifiées. Ces règles sont fournies dans une rubrique spéciale « Règles de typage » lors de la description de chaque prédicat, expression ou substitution dans les chapitres 4, 5 et 6.

3.2 Les types B

Syntaxe

Type ::= *Type_de_base*
| "ℙ" "(" *Type* ")"
| *Type* "×" *Type*
| "struct" "(" (Ident ":" *Type*)$^{+","}$ ")"
| "(" *Type* ")"

Type_de_base ::=
"ℤ"
| "ℝ"
| "FLOAT"
| "BOOL"
| "STRING"
| Ident

Description

Un type B est soit un type de base, soit construit à l'aide d'un constructeur de type.

Les types de base sont :

- l'ensemble des entiers relatifs ℤ,
- l'ensemble des nombres réels ℝ,
- l'ensemble des nombres flottants FLOAT,
- l'ensemble des booléens BOOL, défini par BOOL = {TRUE, FALSE}, avec TRUE ≠ FALSE,

- l'ensemble des chaînes de caractères STRING,
- les ensembles abstraits et les ensembles énumérés introduits dans la clause SETS ainsi que les paramètres ensembles de machine qui sont considérés comme des ensembles abstraits.

Il existe trois constructeurs de types : l'ensemble des parties noté '$\mathbb{P}$', le produit cartésien noté '×' et la collection d'articles (ou *records* en anglais) noté 'struct'. Soient *T*, *T1*, *T2*, *T3*, *T4* des types,

- $\mathbb{P}(T)$ désigne l'ensemble des parties de *T*, c'est-à-dire l'ensemble dont les éléments sont des ensembles d'éléments de *T*,
- $T1 \times T2$ désigne le produit cartésien des ensembles *T1* et *T2*, c'est-à-dire l'ensemble des paires ordonnées dont le premier élément est de type *T1* et le second est de type *T2*. L'opérateur '×' étant associatif à gauche, le type $T1 \times T2 \times T3 \times T4$ désigne en fait le type $((T1 \times T2) \times T3) \times T4$,
- Soient n un entier supérieur ou égal à 1, T_1, …, T_n des types et $Ident_1$, …, $Ident_n$ des identificateurs deux à deux distincts. Alors le type record struct $(Ident_1 : T_1, \ldots, Ident_n : T_n)$ désigne l'ensemble formé par une collection ordonnée de n types T_i appelés champs du record. Chaque champ possède un nom $Ident_i$ appelé label. Les labels d'un type record doivent être deux à deux distincts.

Exemples

Le type de l'expression 3 est $\mathbb{Z}$.

Le type de l'expression $\{-5, 3, -1, 8\}$ est $\mathbb{P}(\mathbb{Z})$.

Le type de l'expression $(0 .. 10) \times$ BOOL $\rightarrow$ ABS1 est $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times$ BOOL $\times$ ABS1$))$.

Le type de l'expression rec (a : 5, b : TRUE) est struct (a : $\mathbb{Z}$, b : BOOL).

3.3 Typage des données abstraites

Syntaxe

Typage_donnée_abstraite ::=
    Ident$^{+","}$ "∈" *Expression*$^{+"\times"}$
  | Ident "⊆" *Expression*
  | Ident "⊂" *Expression*
  | Ident$^{+","}$ "=" *Expression*$^{+","}$

Description

On appelle donnée abstraite une constante abstraite, une variable abstraite ou une donnée introduite par une substitution ANY, LET, par un prédicat ∀ ou ∃, ou par une expression λ, Σ, Π, ⋃, { | } ou ⋂ (cf. §3.1 *Fondements du typage*).

Les prédicats de typage des données abstraites sont des prédicats élémentaires particuliers. Chaque prédicat de typage permet de typer une ou plusieurs données abstraites. Il est séparé des prédicats précédents et suivants par une conjonction.

Les prédicats élémentaires de typage sont l'appartenance, l'inclusion et l'égalité. Les données abstraites à typer doivent se trouver en partie gauche de l'opérateur d'appartenance, d'inclusion ou d'égalité. La partie droite est constituée par une expression dont tous les constituants sont des données accessibles et déjà typées (cf. *Ordre du typage*, ci-dessous). Le type de la donnée abstraite en partie gauche est alors déterminé en appliquant les règles de typage du prédicat utilisé.

Ordre du typage

Le mécanisme du typage des données au sein d'un prédicat consiste à parcourir l'ensemble du texte du prédicat du début jusqu'à la fin. Lorsqu'une donnée qui n'est pas encore typée apparaît en partie gauche d'un prédicat de typage, la donnée devient typée et le reste dans la suite du texte du prédicat.

Exemples

$$
\begin{array}{l}
\mathit{VarRaf1} \in \mathrm{INT} \wedge \\
\mathit{VarRaf2} \subseteq \mathrm{NAT} \wedge \\
\mathit{VarRaf3} = \mathrm{TRUE} \wedge \\
\mathit{VarRaf4} \in (0 .. 5) \leftrightarrow (0 .. 10) \wedge \\
\mathit{VarRaf5} \in \mathbb{Z} \wedge \\
\mathit{VarRaf6} \subseteq \mathrm{NAT} \wedge \\
\mathit{VarRaf7} \subset \mathrm{NAT}_1 \cup (-5 .. -1) \wedge \\
\mathit{VarRaf8} = (0 .. 4) \times \{\mathrm{FALSE}\}
\end{array}
$$

Comme le type de INT est $\mathbb{P}(\mathbb{Z})$ et que *VarRaf1* est typé à l'aide de l'opérateur '$\in$', le type de *VarRaf1* est $\mathbb{Z}$.

Comme le type de NAT est $\mathbb{P}(\mathbb{Z})$ et que *VarRaf2* est typé à l'aide de l'opérateur '$\subseteq$', le type de *VarRaf2* est $\mathbb{P}(\mathbb{Z})$.

Comme le type de TRUE est BOOL et que *VarRaf3* est typé à l'aide de l'opérateur '=', le type de *VarRaf3* est BOOL.

De même, le type de *VarRaf4* est $\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$, le type de *VarRaf5* est $\mathbb{Z}$, le type de *VarRaf6* est $\mathbb{P}(\mathbb{Z})$, le type de *VarRaf7* est $\mathbb{P}(\mathbb{Z})$ et le type de *VarRaf8* est $\mathbb{P}(\mathbb{Z} \times \mathrm{BOOL})$.

3.4 Types et contraintes des données concrètes

Les données concrètes d'un module B sont les données qui feront partie du programme associé au module (cf. §8.2 *Implantation*). Comme les données concrètes doivent pouvoir être implémentées par un programme, certaines contraintes ont été fixées pour différencier les données concrètes de celles qui ne le sont pas. Ces contraintes sont forcément arbitraires, mais elles ont été établies en considérant ce que savent implémenter facilement les langages de programmation comme Ada ou C++ et en essayant de donner un maximum de souplesse aux utilisateurs du Langage B. Ainsi, les entiers les booléens ou les tableaux, pourront être implémentés (éventuellement sous certaines contraintes), mais pas la donnée de valeur $\{-1, 5, 8\}$ car elle n'est pas implémentable directement et simplement dans les langages de programmation classiques.

La plus importante de ces contraintes est le type des données. Par exemple, une donnée de type $\mathbb{P}(\mathbb{Z} \times \mathbb{P}(\mathbb{Z}))$ qui représente un ensemble de couples dont les premiers éléments sont des entiers et dont les seconds éléments sont des ensembles d'entiers n'est pas retenue comme concrète car elle s'éloigne trop de ce qu'un langage de programmation peut implémenter directement.

Il existe d'autres contraintes que le type. Par exemple les seules données entières concrètes sont celles qui sont comprises entre le plus petit entier implémentable et le plus grand entier implémentable pour une machine cible donnée.

Enfin les contraintes sur les données concrètes dépendent de la nature de la donnée. Par exemple les constantes concrètes peuvent être des intervalles d'entiers mais pas les variables concrètes. Nous allons, dans un premier temps, décrire toutes les catégories possibles de données concrètes, puis nous donnerons sous la forme d'un tableau les catégories autorisées, pour chaque nature de donnée concrète.

Les ensembles abstraits ou énumérés

Type

Un ensemble abstrait ou énuméré *Ens* est de type $\mathbb{P}(Ens)$.

Contraintes

Aucune contrainte pour les ensembles énumérés.

Les entiers

Type

Les entiers concrets sont de type $\mathbb{Z}$.

Contraintes

Les entiers concrets doivent appartenir à l'intervalle INT dont les bornes inférieures et supérieures sont les constantes prédéfinies MININT et MAXINT. La valeur de ces constantes est paramétrable, elle dépend de la machine cible, sur lequel le programme associé à un projet B doit fonctionner. Ces valeurs doivent être telles que tout entier compris entre MININT et MAXINT soit représentable directement et sans débordement sur la machine cible en question.

Les flottants

Type

Les nombre flottants sont de type FLOAT.

Contraintes

Aucune contrainte.

Les booléens

Type

Les booléens sont de type BOOL.

Contraintes

aucune contrainte

Les éléments d'ensembles abstraits ou énumérés

Type

Les éléments appartenant à un ensemble abstrait ou énuméré *Ens* sont du type *Ens*.

Contraintes

aucune contrainte

Les sous-ensembles d'entiers ou d'éléments d'ensembles abstraits

Type

Les sous-ensembles concrets d'entiers sont de type $\mathbb{P}(\mathbb{Z})$. Les sous-ensembles concrets d'éléments d'ensemble abstrait *Ens* sont de type $\mathbb{P}(Ens)$.

Contraintes

Les sous-ensembles concrets d'entiers doivent être des intervalles d'entiers concrets. Les sous-ensembles concrets d'éléments d'ensemble abstrait doivent être des intervalles d'entiers concrets, lorsque l'ensemble abstrait est valué par un intervalle d'entiers (cf. §7.17 *La clause VALUES*).

Les tableaux

Type

Les tableaux concrets sont de type $\mathbb{P}(T1 \times \ldots \times Tn \times T_0)$, où $n \geq 1$ et chaque type Ti est un type de base autre que le type STRING.

Contraintes

Avant de définir la notion de tableau concret, nous introduisons la notion d'ensemble simple concret. Un ensemble simple concret est un ensemble abstrait ou énuméré, l'ensemble des booléens ou un intervalle d'entiers concrets ou un sous-ensemble concret d'un ensemble abstrait.

Un tableau concret est une fonction totale dont l'ensemble de départ est le produit cartésien de n ensembles simples concrets (avec $n \geq 1$) et dont l'ensemble d'arrivée est un ensemble simple concret. Les n ensembles simples qui constituent le domaine de définition du tableau s'appellent aussi les ensembles indices du tableau.

Exemple

$$
\begin{aligned}
&Tab1 \in (0 \,..\, 4) \rightarrow \mathrm{INT} \;\wedge \\
&Tab2 \in EnsAbstrait1 \times EnsEnum1 \times \mathrm{BOOL} \twoheadrightarrow \mathrm{BOOL} \;\wedge \\
&Tab3 \in (-1 \,..\, 1) \times CteInterv1 \rightarrowtail (0 \,..\, 100) \;\wedge \\
&Tab4 \in EnsEnum1 \rightarrowtail\!\!\!\!\twoheadrightarrow \mathrm{NAT} \;\wedge \\
&Tab5 \in (0 \,..\, 8) \rightarrow \mathrm{INT}
\end{aligned}
$$

Le tableau concret *Tab1* est une fonction totale de l'intervalle $(0 \,..\, 4)$ dans l'ensemble INT. Le tableau concret *Tab2* est une surjection totale du produit cartésien des ensembles simples *EnsAbstrait1*, *EnsEnum1* et BOOL dans l'ensemble BOOL.

Les records

Type

Le type des records concrets se définit par induction. Un type de records concret est un type record dont chaque champ est de l'un des types suivants : $\mathbb{Z}$, BOOL, un ensemble abstrait, un ensemble énuméré, un type de tableau concret ou un type de record concret.

Contraintes

Les contraintes sur les données records concrètes se définissent par induction. Chaque champ d'un record concret doit être une donnée concrète. En particulier, si l'un des champs d'un record concret est lui-même un record, alors chaque champ de ce dernier doit à nouveau être une donnée concrète.

Exemple

$$
\begin{array}{lll}
Annee \in \text{struct} (& An : \text{INT}, & \\
 & Bissextile : \text{BOOL}, & \\
 & NbrJours : (1 .. 12) \rightarrow (28 .. 31), & \\
 & Meteo : \text{struct} (& TempMoy : (1 .. 12) \rightarrow \text{INT}, \\
 & & PrecipMoy : (1 .. 12) \rightarrow \text{INT}))
\end{array}
$$

Le record concret *Annee* contient quatre champs : le champ *An* est un entier concret, le champ *Bissextile* est un booléen, le champ *NbrJours* est un tableau concret et le champ *Meteo* est un record concret possédant deux champs tableaux concrets, *TempMoy* et *PrecipMoy*.

Les chaînes de caractères

Type

Les chaînes de caractères sont de type STRING.

Contraintes

aucune contrainte

Tableau récapitulatif

Le tableau suivant récapitule pour chaque nature de donnée concrète, quels sont les types de données concrètes autorisés.

Type concret / Nature	Ens. abstrait ou énuméré	Entier	Booléen	Élément d'ens. abstrait ou énuméré	Intervalle d'entiers ou sous-ensemble d'ens. abstrait	Tableau	Record	Chaîne de caractères
Paramètre de machine (ensemble)	×				×			
Ensemble abstrait ou énuméré	×							
Paramètre de machine (scalaire)		×	×	×				
Énuméré littéral				×				
Constante concrète		×	×	×	×	×	×	
Variable concrète		×	×	×		×	×	
Paramètre d'entrée d'opération (non locale ou locale)		×	×	×		×	×	×
Paramètre de sortie d'opération (non locale ou locale)		×	×	×		×	×	

Type concret / Nature	Ens. abstrait ou énuméré	Entier	Booléen	Élément d'ens. abstrait ou énuméré	Intervalle d'entiers ou sous-ensemble d'ens. abstrait	Tableau	Record	Chaîne de caractères
Variable locale		×	×	×		×	×	

3.5 Typage des constantes concrètes

Syntaxe

```
Typage_cte_concrète ::=
        Ident^(+",")  "∈"  Typage_appartenance_donnée_concrète^(+"×")
   |    Ident  "="  Typage_égalité_cte_concrète
   |    Ident  "⊆"  Ensemble_simple
   |    Ident  "⊂"  Ensemble_simple

Typage_appartenance_donnée_concrète ::=
        Ensemble_simple
   |    Ensemble_simple^(+"×")  "→"  Ensemble_simple
   |    Ensemble_simple^(+"×")  "↣"  Ensemble_simple
   |    Ensemble_simple^(+"×")  "↠"  Ensemble_simple
   |    Ensemble_simple^(+"×")  "⤖"  Ensemble_simple
   |    "{" Terme_simple^(+",") "}"
   |    "struct"  "(" (Ident ":" Typage_appartenance_donnée_concrète)^(+",") ")"

Typage_égalité_cte_concrète ::=
   |    Terme
   |    Expr_tableau
   |    Intervalle_B0
   |    Ensemble_nombres_B0
   |    "rec" "(" ( [ Ident ":" ] ( Terme | Expr_tableau ) )^(+",") ")"

Ensemble_simple ::=
        Ensemble_nombres_B0
   |    "FLOAT"
   |    "BOOL"
   |    Intervalle_B0
   |    Ident

Ensemble_nombres_B0 ::=
        "NAT"
   |    "NAT1"
   |    "INT"

Expr_tableau ::=
        Ident
   |    "{" ( Terme_simple^(+"↦") "↦" Terme )^(+",") "}"
   |    Ensemble_simple^(+"×")  "×"  "{" Terme "}"

Intervalle_B0     ::=
        Expression_arithmétique  ".."  Expression_arithmétique
   |    Ensemble_nombres_B0
```

Description

Les constantes concrètes sont typées à l'aide de prédicats de typage utilisés dans les clauses PROPERTIES. Les prédicats de typage des constantes concrètes suivent les mêmes principes que les prédicats de typage des données abstraites (cf. §3.3 *Typage des données abstraites*), auxquels ils apportent un certain nombre de restrictions : la partie droite du prédicat doit permettre de donner à chaque constante un type de constante concrète (cf. §3.4 *Types et contraintes des données concrètes*). Seules les expressions élémentaires dont la syntaxe est donnée ci-dessus sont autorisées. Il s'agit de :

- l'appartenance à un ensemble simple, qui est un ensemble de scalaires.

  Exemple

  $$Cte1 \in \mathsf{INT} \wedge$$
  $$Cte2 \in \mathsf{BOOL} \wedge$$
  $$Cte3 \in 0 .. 5 \wedge$$
  $$Cte4 \in CteInterv1 \wedge$$
  $$Cte5 \in EnsAbstrait1 \wedge$$
  $$Cte6 \in EnsEnum1$$

  Ces prédicats de typage peuvent également s'écrire sous la forme :

  $$Cte1, Cte2, Cte3, Cte4, Cte5, Cte6 \in \mathsf{INT} \times \mathsf{BOOL} \times (0 .. 5) \times CteInterv1 \times EnsAbstrait1 \times EnsEnum1$$

- l'appartenance à un ensemble de fonctions totales. Les ensembles indices et l'ensemble d'arrivée de la fonction doivent être des ensembles simples.

  Exemple

  $$Tab1 \in (0 .. 4) \rightarrow \mathsf{INT} \wedge$$
  $$Tab2 \in EnsAbstrait1 \times EnsEnum1 \times \mathsf{BOOL} \twoheadrightarrow \mathsf{BOOL} \wedge$$
  $$Tab3 \in (-1 .. 1) \times CteInterv1 \rightarrowtail (0 .. 100) \wedge$$
  $$Tab4 \in EnsEnum1 \rightarrowtail\!\!\!\!\twoheadrightarrow \mathsf{NAT}$$

- l'appartenance à un ensemble en extension de scalaires.

  Exemple

  $$Cte7 \in \{ 0, 3, 7, -8 \} \wedge$$
  $$Cte8 \in \{ bleu, blanc, rouge \}$$

- l'égalité avec un identificateur qui désigne une donnée scalaire.

  Exemple

  $$Cte10 = rouge \wedge$$
  $$Cte11 = Cte10$$

- l'égalité avec une expression arithmétique ou booléenne.

  Exemple

  $$Cte12 = CteInt1 + 2 \times (CteInt2 - 1) \wedge$$
  $$Cte13 = 0 \wedge$$
  $$Cte14 = \mathsf{FALSE} \wedge$$
  $$Cte15 = \mathsf{bool} (Cte12 < 10 \vee Cte12 > 20)$$

- l'égalité avec une expression tableau.

  Exemple

  $$Tab5 = (0 .. 4) \times \{ 0 \} \wedge$$
  $$Tab6 = (0 .. 2) \times \{ \mathsf{TRUE} \}$$

- l'inclusion dans un ensemble simple.

  Exemple

$CteInterv16 \subseteq \mathsf{INT} \wedge$
$CteInterv17 \subseteq EnsAbstrait1 \wedge$
$CteInterv18 \subset -100 .. 100 \wedge$
$CteInterv19 \subset CteInterv16$

- l'appartenance à un ensemble de records.

Exemple

$CteRec \in \mathsf{struct}\ (masc : \mathsf{BOOL}, age : 0 .. 255)$

3.6 Typage des variables concrètes

Syntaxe

Typage_var_concrète ::=
  Ident$^{+","}$ "∈" *Typage_appartenance_donnée_concrète*$^{+"\times"}$
| Ident "=" *Terme*

Description

Les variables concrètes sont typées à l'aide de prédicats de typage utilisés dans les clauses INVARIANT. Les prédicats de typage des variables concrètes suivent les mêmes principes que les prédicats de typage des données abstraites (cf. §3.3 *Typage des données abstraites*) auxquels ils apportent un certain nombre de restrictions : seuls les prédicats d'appartenance et d'égalité sont autorisés pour typer les variables concrètes. L'inclusion est interdite car une variable concrète ne peut pas être un ensemble. De plus, la partie droite du prédicat de typage doit permettre de donner à la donnée un type de variable concrète (cf. §3.4 *Types et contraintes des données concrètes*). Seules les expressions élémentaires dont la syntaxe est donnée ci-dessus sont autorisées. Il s'agit :

- de l'appartenance à un ensemble simple (qui est un ensemble de scalaires).

Exemple

$Var1 \in \mathsf{INT} \wedge$
$Var2 \in \mathsf{BOOL} \wedge$
$Var3 \in 0 .. 5 \wedge$
$Var4 \in CteInterv1 \wedge$
$Var5 \in EnsAbstrait1 \wedge$
$Var6 \in EnsEnum1$

Ces prédicats de typage peuvent également s'écrire sous la forme :

$Var1, Var2, Var3, Var4, Var5, Var6 \in$
$\mathsf{INT} \times \mathsf{BOOL} \times (0 .. 5) \times CteInterv1 \times EnsAbstrait1 \times EnsEnum1$

- de l'appartenance à un ensemble de fonctions totales. Les ensembles indices et l'ensemble d'arrivée de la fonction doivent être des ensembles simples.

Exemple

$Tab1 \in (0 .. 4) \rightarrow \mathsf{INT} \wedge$
$Tab2 \in EnsAbstrait1 \times EnsEnum1 \times \mathsf{BOOL} \twoheadrightarrow \mathsf{BOOL} \wedge$
$Tab3 \in (-1 .. 1) \times CteInterv1 \rightarrowtail (0 .. 100) \wedge$
$Tab4 \in EnsEnum1 \rightarrowtail\!\!\!\!\twoheadrightarrow \mathsf{NAT}$

- de l'appartenance à un ensemble en extension de scalaires.

Exemple

Var7 $\in$ {0, 3, 7, -8} $\wedge$
Var8 $\in$ {*bleu*, *blanc*, *rouge*}

- de l'égalité avec un identificateur qui désigne une donnée scalaire.

Exemple

Var10 = *rouge* $\wedge$
Var11 = *Var10*

- de l'égalité avec une expression arithmétique ou booléenne.

Exemple

Var12 = *CteInt1* + 2 × (*CteInt2* - 1) $\wedge$
Var13 = 0 $\wedge$
Var14 = FALSE $\wedge$
Var15 = bool (*Var12* < 10 $\vee$ *Var12* > 20)

- de l'appartenance à un ensemble de records.

Exemple

Var16 $\in$ struct (*val* : INT, *ok* : BOOL)

3.7 Typage des paramètres d'entrée d'opération

Syntaxe

Typage_param_entrée ::=
    Ident$^{+","}$ "∈" *Typage_appartenance_param_entrée*$^{+"\times"}$
  | Ident "=" *Terme*

Typage_appartenance_param_entrée ::=
    Ensemble_simple
  | *Ensemble_simple*$^{+"\times"}$ "→" *Ensemble_simple*
  | *Ensemble_simple*$^{+"\times"}$ "↣" *Ensemble_simple*
  | *Ensemble_simple*$^{+"\times"}$ "↠" *Ensemble_simple*
  | *Ensemble_simple*$^{+"\times"}$ "⤖" *Ensemble_simple*
  | "{" *Terme_simple*$^{+","}$ "}"
  | "struct" "(" (Ident ":" *Typage_appartenance_donnée_concrète*)$^{+","}$ ")"
  | "STRING"

Description

Les paramètres d'entrée d'opération (non locale ou locale) sont typés à l'aide de prédicats de typage utilisés dans une précondition (cf. §7.23 *La clause OPERATIONS*).

On distingue deux cas selon que l'opération appartient à un module avec un code associé (module développé ou module de base) ou à un module abstrait (cf. §8.2 *Module B*).

Dans le premier cas, les prédicats de typage des paramètres d'entrée d'opération reprennent les mêmes principes que les prédicats de typage des variables concrètes et ajoutent une nouvelle possibilité. Il s'agit de l'appartenance à l'ensemble des chaînes de caractères STRING. Un paramètre d'entrée d'opération peut donc être une chaîne de caractères.

Exemple

Var17 $\in$ STRING

Dans le cas d'une opération d'un module abstrait, les paramètres d'entrée d'opération sont des données abstraites (cf. §3.3 *Typage des données abstraites*). En effet, un module abstrait n'ayant pas de code associé, ses opérations ne peuvent pas être appelées par une implantation et donc il n'est pas nécessaire que ses paramètres d'entrée d'opération soient concrets.

3.8 Typage des paramètres de machines

Syntaxe

Typage_param_mch ::=
    Ident$^{+\text{","}}$ "∈" *Typage_appartient_param_mch*$^{+\text{"×"}}$
| Ident$^{+\text{","}}$ "=" *Terme*$^{+\text{","}}$

Typage_appartient_param_mch ::=
    Ensemble_nombres
| "BOOL"
| *Intervalle*
| Ident

Ensemble_nombres ::=
    "ℤ"
| "ℕ"
| "$\mathbb{N}_1$"
| "NAT"
| "NAT_1"
| "INT"

Description

Les *paramètres* scalaires d'une machine abstraite sont typés à l'aide de prédicats de typage dans la clause CONSTRAINTS (cf. §7.5 La clause CONSTRAINTS). Les prédicats de typage des paramètres scalaires d'une machine suivent les mêmes principes que les prédicats de typage des données abstraites (cf. §3.3 *Typage des données abstraites*), auxquels ils apportent un certain nombre de restrictions.

Les seuls types autorisés pour typer explicitement les paramètres scalaires d'une machine sont ℤ, BOOL et les paramètres ensembles de la machine abstraite. Les paramètres ensembles sont exactement les paramètres représentés par des identificateurs sans caractères minuscules.

Les paramètres formels scalaires déjà typés dans un prédicat de typage, peuvent servir à typer un autre paramètre formel scalaire.

3.9 Typage des variables locales et des paramètres de sortie d'opération

Description

Les variables locales déclarées dans une substitution VAR et les paramètres formels de sortie d'opération (non locale ou locale) sont typés au moyen de substitutions de typage (cf. chapitre 6 *Substitutions*). Les substitutions de typage sont les substitutions « devient égal » (cf. §6.3), « devient élément de » (cf. §6.12), « devient tel que » (cf. §6.13) et appel d'opération (cf. §6.16).

En ce qui concerne les paramètres de sortie d'opération, on distingue deux cas selon que l'opération appartient à un module avec un code associé (module développé ou module de base) ou à un module abstrait (cf. §8.2 *Module B*).

Dans le cas d'une opération d'un module abstrait, les paramètres de sortie d'opération sont des données abstraites. En effet, un module abstrait n'ayant pas de code associé, ses opérations ne peuvent pas être appelées par une implantation et donc il n'est pas nécessaire que ses paramètres de sortie d'opération soient concrets.

Les paramètres de sortie des modules avec code associé ainsi que les variables locales d'implantations doivent être des données concrètes. Soit *v1* une donnée désignant une variable locale ou un paramètre de sortie d'opération non encore typée (cf. §3.3 *Ordre du typage*). Voici comment *v1* peut être typée à l'aide des différentes substitutions de typage.

Substitution « devient égal »

Soit E une expression de type T, alors la substitution « devient égal » : $v1 := E$, donne à *v1* le type T. Pour que E puisse servir à typer *v1*, il faut que la variable *v1* n'apparaisse pas dans E. Il est également possible de typer *v1* dans une substitution « devient égal » portant sur plusieurs données en parallèle.

Exemple

$v1 :=$ TRUE	*v1* est de type BOOL
$v1, v2 := 0, 0$	*v1* et *v2* sont de type $\mathbb{Z}$

Substitution « devient élément »

Soit E une expression de type $\mathbb{P}(T)$, alors la substitution « devient élément de » : $v1 :\in E$, donne à *v1* le type T.

Exemple

$v1 :\in AbsSet$	*v1* est de type *AbsSet*

Substitution « devient tel que »

Soit P un prédicat, alors la substitution « devient tel que » : $v1 :(P)$, doit typer la donnée *v1* à l'aide d'un prédicat de typage de donnée abstraite, selon les principes décrits au §3.3.

Exemple

$v1 :(v1 \in \mathsf{INT} \wedge v1 < 10)$	*v1* est de type $\mathbb{Z}$

Substitution « appel d'opération »

Enfin, *v1* peut être typé comme paramètre de sortie effectif d'un appel d'opération (non locale ou locale). Le type de *v1* est alors donné par le type du paramètre de sortie de l'opération appelée.

Exemple

$v1 \longleftarrow$ op1	*v1* est du type du paramètre de sortie de op1

4 PREDICATS

Syntaxe

Prédicat ::=		
	Prédicat_parenthésé	Propositions
|	*Prédicat_conjonction*	
|	*Prédicat_négation*	
|	*Prédicat_disjonction*	
|	*Prédicat_implication*	
|	*Prédicat_équivalence*	
|	*Prédicat_universel*	Prédicats quantifiés
|	*Prédicat_existentiel*	
|	*Prédicat_égalité*	Prédicats d'égalité
|	*Prédicat_inégalité*	
|	*Prédicat_appartenance*	Prédicats d'appartenance
|	*Prédicat_non_appartenance*	
|	*Prédicat_inclusion*	Prédicats d'inclusion
|	*Prédicat_inclusion_stricte*	
|	*Prédicat_non_inclusion*	
|	*Prédicat_non_inclusion_stricte*	
|	*Prédicat_inférieur_ou_égal*	Prédicat de comparaison d'entiers
|	*Prédicat_strictement_inférieur*	
|	*Prédicat_supérieur_ou_égal*	
|	*Prédicat_strictement_supérieur*	

Description

Un prédicat est une formule qui peut être prouvée ou réfutée, ou qui peut faire partie des hypothèses sous lesquelles on fait une preuve.

Les prédicats sont utilisés dans le langage B pour :

- exprimer les propriétés de données (au sein des clauses CONSTRAINTS, PROPERTIES, INVARIANT, ASSERTIONS, des prédicats $\forall$ ou $\exists$, des expressions λ, $\{\,|\,\}$, Σ, Π, $\bigcup$ ou $\bigcap$ et des substitutions « devient tel que », ANY, LET, PRE, ASSERT ou WHILE),
- exprimer des conditions lors de l'application de substitutions (substitutions SELECT, IF, WHILE).

Les sections suivantes décrivent les prédicats regroupés par familles. Pour une famille de prédicats, on présente successivement l'opérateur du prédicat, sa syntaxe dans une notation mathématique, ses règles de typage, les éventuelles règles de portée des données qui y sont déclarées, ses restrictions sémantiques, sa description, certaines lois ou propriétés mathématiques et enfin des exemples.

4.1 Propositions

Opérateur

( )	Parenthèses
$\land$	Conjonction
$\neg$	Négation
$\lor$	Disjonction
$\Rightarrow$	Implication
$\Leftrightarrow$	Équivalence

Syntaxe

Prédicat_parenthésé	::=	"(" *Prédicat* ")"
Prédicat_conjonction	::=	*Prédicat* "$\land$" *Prédicat*
Prédicat_négation	::=	"$\neg$" "(" *Prédicat* ")"
Prédicat_disjonction	::=	*Prédicat* "$\lor$" *Prédicat*
Prédicat_implication	::=	*Prédicat* "$\Rightarrow$" *Prédicat*
Prédicat_équivalence	::=	*Prédicat* "$\Leftrightarrow$" *Prédicat*

Définition

$P \lor Q \;\hat{=}\; \neg(P) \Rightarrow Q$

$P \Leftrightarrow Q \;\hat{=}\; (P \Rightarrow Q) \land (Q \Rightarrow P)$

Description

Les opérateurs présentés permettent la construction de prédicats complexes à partir de prédicats plus simples. On donne ci-dessous, pour chaque prédicat complexe, la liste complète des cas pour lesquels le prédicat est vrai :

Soient P et Q des prédicats,

- (P) est vrai si et seulement si P est vrai. Cette construction permet de parenthéser les prédicats, ce qui peut se révéler nécessaire selon la priorité des opérateurs utilisés.
  Par exemple, le prédicat $P \land Q \Rightarrow \mathrm{R}$ sera analysé comme $(P \land Q) \Rightarrow \mathrm{R}$ et non pas comme $P \land (Q \Rightarrow \mathrm{R})$, car l'opérateur '$\land$' est plus prioritaire que l'opérateur '$\Rightarrow$'. Pour exprimer le prédicat $P \land (Q \Rightarrow \mathrm{R})$ les parenthèses sont donc obligatoires.
- $P \land Q$ est vrai si et seulement si P et Q sont vrais,
- $\neg(P)$ est vrai si et seulement si P n'est pas vrai,
- $P \lor Q$ est vrai si et seulement si P ou Q sont vrais,
- $P \Rightarrow Q$ est vrai si et seulement si Q est vrai ou P n'est pas vrai,
- $P \Leftrightarrow Q$ est vrai si et seulement si $P \Rightarrow Q$ et $Q \Rightarrow P$ sont vrais.

4.2 Prédicats quantifiés

Opérateur

$\forall$ Quantificateur universel

$\exists$ Quantificateur existentiel

Syntaxe

Prédicat_universel ::= "$\forall$" *Liste_ident* "." "(" *Prédicat* "$\Rightarrow$" *Prédicat* ")"

Prédicat_existentiel ::= "$\exists$" *Liste_ident* "." "(" *Prédicat* ")"

Définition

$\exists X . (P) \ \hat{=}\ \neg (\forall X . (\neg (P)))$

Règle de portée

Les prédicats $\forall X . (P)$ et $\exists X . (P)$ introduisent la déclaration d'une liste de données X dont la portée est le prédicat P.

Restrictions

1. Les variables introduites par un prédicat universel de la forme $\forall X . (P \Rightarrow Q)$ doivent être typées par un prédicat de typage de données abstraites (cf. §3.3 *Typage des données abstraites*), dans une liste de conjonctions situées au plus haut niveau d'analyse syntaxique de P. Ces variables ne peuvent pas être utilisées dans P avant d'avoir été typées.
2. Les variables introduites par un prédicat existentiel de la forme $\exists X . (P)$ doivent être typées par un prédicat de typage de données abstraites (cf. §3.3 *Typage des données abstraites*), dans une liste de conjonctions situées au plus haut niveau d'analyse syntaxique de P. Ces variables ne peuvent pas être utilisées dans P avant d'avoir été typées.

Description

Soit X une liste d'identificateurs deux à deux distincts et P et Q des prédicats.

- Le prédicat $\forall X . (P \Rightarrow Q)$ est vrai si le prédicat $P \Rightarrow Q$ est vrai quelles que soient les valeurs de X.
- Le prédicat $\exists X . (P)$ est vrai s'il existe un ensemble non vide de valeurs pour X pour lesquelles le prédicat P est vrai.

Exemples

Soit l'ensemble d'entiers : $A = \{ 0, 1, 2 \}$.

Le prédicat $\forall x . (x \in A \Rightarrow x \leq 2)$ est vrai puisque tout élément de A est inférieur ou égal à 2.

Le prédicat $\exists x . (x \in A \wedge x \leq 1)$ est vrai puisqu'il existe un élément de A inférieur ou égal à 1.

4.3 Prédicats d'égalité

Opérateur

$=$	Égalité
$\neq$	Inégalité

Syntaxe

Prédicat_égalité	::=	*Expression* "=" *Expression*
Prédicat_inégalité	::=	*Expression* "≠" *Expression*

Règle de typage

Dans les prédicats $x = y$ et $x \neq y$, les expressions x et y doivent avoir le même type.

Définition

$x \neq y \;\;\hat{=}\;\; \neg\,(x = y)$

Description

- Le prédicat $x = y$ est vrai si et seulement si les expressions x et y ont la même valeur.
- Le prédicat $x \neq y$ est vrai si et seulement si les expressions x et y n'ont pas la même valeur.

4.4 Prédicats d'appartenance

Opérateur

$\in$	Appartenance
$\notin$	Non appartenance

Syntaxe

Prédicat_appartenance	::=	*Expression* "$\in$" *Expression*
Prédicat_non_appartenance	::=	*Expression* "$\notin$" *Expression*

Règle de typage

Dans les prédicats $x \in E$ et $x \notin E$, si le type de l'expression x est T alors le type de E doit être $\mathbb{P}(T)$.

Définition

$x \notin E \ \widehat{=}\ \neg\,(x \in E)$

Description

Soient x et E des expressions.

- Le prédicat $x \in E$ est vrai si et seulement si la valeur de l'expression x appartient à l'ensemble E.
- Le prédicat $x \notin E$ est vrai si et seulement si la valeur de l'expression x n'appartient pas à l'ensemble E.

4.5 Prédicats d'inclusion

Opérateur

$\subseteq$	Inclusion
$\subset$	Inclusion stricte
$\not\subseteq$	Non inclusion
$\not\subset$	Non inclusion stricte

Syntaxe

Prédicat_inclusion	::=	*Expression* "$\subseteq$" *Expression*
Prédicat_inclusion_stricte	::=	*Expression* "$\subset$" *Expression*
Prédicat_non_inclusion	::=	*Expression* "$\not\subseteq$" *Expression*
Prédicat_non_inclusion_stricte	::=	*Expression* "$\not\subset$" *Expression*

Définitions

$s \subseteq T \;\hat{=}\; s \in \mathbb{P}(T)$

$s \subset T \;\hat{=}\; s \subseteq T \wedge s \neq T$

$s \not\subseteq T \;\hat{=}\; \neg\,(s \in \mathbb{P}(T))$

$s \not\subset T \;\hat{=}\; \neg\,(s \subseteq T \wedge s \neq T)$

Règle de typage

Dans les prédicats $X \subseteq Y$, $X \subset Y$, $X \not\subseteq Y$, $X \not\subset Y$, les expressions X et Y sont du même type, et leur type est de la forme $\mathbb{P}(T)$.

Description

Soient X et Y des expressions représentant des ensembles.

- $X \subseteq Y$ est vrai si tout élément de X appartient à Y.
- $X \subset Y$ est vrai si tout élément de X appartient à Y et si X est différent de Y.
- $X \not\subseteq Y$ est vrai s'il existe un élément de X qui n'appartient pas à Y.
- $X \not\subset Y$ est vrai si X est égal à Y ou s'il existe un élément de X qui n'appartient pas à Y.

4.6 Prédicats de comparaison de nombres

Opérateur

$\leq$	Inférieur ou égal
$<$	Strictement inférieur
$\geq$	Supérieur ou égal
$>$	Strictement supérieur

Syntaxe

Prédicat_inférieur_ou_égal	::=	*Expression* "≤" *Expression*
Prédicat_strictement_inférieur	::=	*Expression* "<" *Expression*
Prédicat_supérieur_ou_égal	::=	*Expression* "≥" *Expression*
Prédicat_strictement_supérieur	::=	*Expression* ">" *Expression*

Règle de typage

Dans les prédicats $x \leq y$, $x < y$, $x \geq y$, $x > y$, les expressions x et y doivent être de type $\mathbb{Z}$, $\mathbb{R}$ ou FLOAT. Les deux opérandes doivent être du même type.

Description

Soient x et y des expressions représentant des nombres :

- $x \leq y$ est vrai si x est inférieur ou égal à y,
- $x < y$ est vrai si x est strictement inférieur à y,
- $x \geq y$ est vrai si x est supérieur ou égal à y,
- $x > y$ est vrai si x est strictement supérieur à y.

5 EXPRESSIONS

Syntaxe

Expression ::=
Expression_primaire
| *Expression_booléenne*
| *Expression_arithmétique*
| *Expression_de_couples*
| *Expression_d_ensembles*
| *Construction_d_ensembles*
| *Expression_de_records*
| *Expression_de_relations*
| *Expression_de_fonctions*
| *Construction_de_fonctions*
| *Expression_de_suites*
| *Construction_de_suites*
| *Expression_d_arbres*

Description

Une expression est une formule qui désigne une donnée. Une expression possède une valeur qui appartient à un type du langage B.

Les sections suivantes décrivent les expressions regroupées par familles. Pour une famille d'expressions, on donne le nom des expressions, leur syntaxe dans une notation mathématique, leurs règles de typage, les éventuelles règles de portée des données qui y sont déclarées, leur définition, leurs éventuelles conditions de bonne définition mathématique, leur description et des exemples.

5.1 Expressions primaires

Opérateur

.	Renommage d'une donnée
$0	Valeur précédente (dollar 0)
( )	Expression parenthésée
" "	Chaîne de caractères

Syntaxe

Donnée	::=	*Ident_ren*
	|	*Ident_ren*"$0"
Expr_parenthésée	::=	"(" *Expression* ")"
Chaîne_lit	::=	Chaîne_de_caractères

Règles de typage

- Soit *d* le nom d'une donnée, de type *T* et *r* un préfixe de renommage. Alors, le type de *r.d*, *d*$0 et *r.d*$0 est *T*.
- Soit *E* une expression de type *T*, le type de (*E*) est *T*.
- Une chaîne de caractères littérale est de type STRING.

Restrictions

1. L'expression *d* désigne une donnée définie dans un composant B. Il peut s'agir d'un paramètre formel du composant, d'un ensemble abstrait ou énuméré, d'un élément d'un ensemble énuméré, d'une variable, d'une constante, d'un paramètre formel d'opération, d'une donnée abstraite (introduite par un prédicat de quantification ou par une substitution ANY ou LET) ou d'une variable locale (introduite par une substitution VAR).
2. L'expression *r.d*, où *r* est une suite d'identificateurs séparés par des points, désigne une donnée qui à l'origine est déclarée dans un autre composant sous le nom *d*. Le préfixe *r*, dénote les renommages successifs que *d* subit lors de l'*inclusion* ou de l'*importation* d'instances de machines renommées (cf. §8.3 *Instanciation et renommage*).
3. L'expression *d*$0 ou dans le cas général *r.d*$0 ne peut se rencontrer que dans l'un de ces deux cas :
   - dans le prédicat d'une substitution « devient tel que » (cf. §6.13 *Substitution devient tel que*) : alors *d* doit faire partie de la liste des variables de la substitution « devient tel que »,
   - dans l'invariant d'une substitution « boucle tant que » (cf. §6.17 *Substitution boucle tant que*) : alors *d* doit désigner une variable de l'abstraction du composant, implantée par homonymie avec une variable concrète de l'implantation ou bien implantée par homonymie avec une variable d'une machine *importée*.
4. Une chaîne de caractères littérale ne peut se rencontrer qu'en tant que paramètre effectif d'entrée d'un appel d'opération d'une machine de base.

Description

- L'expression *Ident_ren* désigne une donnée *d* définie dans un composant B. Il peut s'agir d'un paramètre formel du composant, d'un ensemble abstrait ou énuméré, d'un élément d'un ensemble énuméré, d'une variable, d'une constante, d'un paramètre formel d'opération, d'une donnée abstraite (introduite par un prédicat de quantification ou par une substitution ANY ou LET) ou d'une variable locale (introduite par une substitution VAR). Lorsque le nom de la donnée *d* comporte des préfixes, ceux-ci dénotent les renommages successifs de *d* (cf. §8.3 *Instanciation et renommage*).
- Soit une donnée *d*. L'expression *d*$0 ne peut se rencontrer que dans les deux cas suivants :
  - dans le prédicat d'une substitution « devient tel que » portant sur la donnée *d*, *d*$0 désigne la valeur de *d* avant l'application de la substitution (cf. §6.13 *Substitution devient tel que*),
  - dans l'invariant d'une substitution « tant que », si *d* est une variable de l'abstraction de l'implantation, *d*$0 désigne la valeur de la variable *d* de l'abstraction avant l'appel de l'opération dans laquelle se situe la substitution « tant que » (cf. §6.17 *Substitution boucle tant que*).
- Une expression entre parenthèses est égale à l'expression située à l'intérieur de ces parenthèses. L'emploi des parenthèses est parfois rendu obligatoire pour représenter certaines expressions, car l'analyse syntaxique d'une expression dépend de la priorité et de l'associativité des opérateurs concernés.
- Une chaîne de caractères littérale est une suite de caractères délimitée par des guillemets ' " ' (cf. §2.1 *Conventions lexicales*). L'utilisation des chaînes de caractères littérales est très restreinte dans le Langage B. La seule utilisation permise consiste à passer une chaîne de caractères littérale en paramètre d'entrée d'une opération de machine de base, afin de communiquer des messages textuels au code associé à la machine de base, qui lui est capable de manipuler les chaînes de caractères.

Exemples

Noms de données : *x1*, *Lundi*, *nbr_de_jours*, *a1.b1.CTE_DEBUT*

Noms de données avant substitutions, ou bien données de l'abstraction : *x1*$0, *cc_02.var*$0

Expressions parenthésées : $(x+y) \times z$, l'expression $x+y$ doit être parenthésée car l'opérateur '×' est plus prioritaire que l'opérateur '+'.

Chaîne de caractères littérale : "Hello world!"

5.2 Expressions booléennes

Opérateur

TRUE	valeur vraie
FALSE	valeur fausse
bool	conversion d'un prédicat en expression booléenne

Syntaxe

Booléen_lit ::= "FALSE"
| "TRUE"

Conversion_bool ::= "bool" "(" *Prédicat* ")"

Règle de typage

Le type des expressions booléennes est BOOL.

Définition

BOOL ≙ { FALSE, TRUE }

Description

- TRUE et FALSE sont les constantes littérales de l'ensemble prédéfini BOOL (cf. §3.2 *Les types B*).
- L'opérateur bool permet de convertir un prédicat en une expression booléenne. Soit P un prédicat, l'expression bool (P) prend la valeur TRUE si P est vrai et FALSE sinon.

Exemples

L'expression : bool ($\exists x . (x \in \mathbb{N}_1 \wedge x = x^2)$) a pour valeur TRUE.

L'expression : bool (b = TRUE) a pour valeur b.

5.3 Expressions arithmétiques

Opérateur

MAXINT	Plus grand entier implémentable
MININT	Plus petit entier implémentable
+	Addition
-	Différence, et aussi moins unaire
×	Produit
/	Division
mod	Modulo
x^y	Puissance
succ	Successeur
pred	Prédécesseur
floor	Partie entière
ceiling	Partie entière par excès
real	Conversion de ℤ dans ℝ

Syntaxe

Entier_lit	::=	Entier_littéral
	|	"MAXINT"
	|	"MININT"
Addition	::=	*Expression* "+" *Expression*
Différence	::=	*Expression* "-" *Expression*
Moins_unaire	::=	"-" *Expression*
Produit	::=	*Expression* "×" *Expression*
Division	::=	*Expression* "/" *Expression*
Modulo	::=	*Expression* "mod" *Expression*
Puissance	::=	$\mathit{Expression}^{\mathit{Expression}}$
Successeur	::=	"succ" ["(" *Expression* ")"]
Prédécesseur	::=	"pred" ["(" *Expression* ")"]
Partie_entière	::=	"floor" "(" *Expression* ")"
Partie_entière_par_excès	::=	"ceiling" "(" *Expression* ")"
Conversion_ℤ⊄ℝ	::=	"real" "(" *Expression* ")"

Règles de typage

Les entiers littéraux ainsi que les constantes prédéfinies MAXINT et MININT sont de type ℤ. Les réels littéraux sont de type ℝ. Dans les expressions : $x + y$, $x - y$, $- x$, $x \times y$, x / y et x^y les expressions x et y doivent être de type ℤ, ℝ ou FLOAT. Dans les expressions : x mod y, x^y, succ (x) et pred (x), les expressions x et y doivent être de type ℤ. Le type de ces expressions est le même que celui des opérandes. Le type des fonctions successeur et prédécesseur est ℙ(ℤ × ℤ).

Dans les expressions floor(x) et ceiling(x), x doit être de type $\mathbb{R}$. Le type des fonctions floor et ceiling est $\mathbb{P}(\mathbb{R} \times \mathbb{Z})$.

Dans l'expression real(x), x doit être de type $\mathbb{Z}$. Le type de la fonction real est $\mathbb{P}(\mathbb{Z} \times \mathbb{R})$.

Bonne définition

Expression	Condition de bonne définition
a / b	$b \in \mathbb{Z} - \{0\}$
a mod b	$a \in \mathbb{N} \wedge b \in \mathbb{N}_1$
a^b	$a \in \mathbb{Z} \wedge b \in \mathbb{N}$

Restriction

1. Un entier littéral doit être compris entre MININT et MAXINT.

Description

Soient x et y des expressions de type $\mathbb{Z}$, alors :

- Les constantes prédéfinies MAXINT et MININT représentent respectivement le plus grand et le plus petit entier concret utilisable en B. Les entiers littéraux dans le langage B doivent être compris entre MININT et MAXINT. Les valeurs de MAXINT et MININT sont fixées pour un projet donné en fonction de la machine cible sur laquelle s'exécutera le programme. Si les entiers sont stockés sur une certaine machine sur quatre octets, alors les valeurs de MAXINT et MININT pourront être 2^{31}-1 et -2^{31}.
- $x + y$ représente la somme de x et y.
- $x - y$ représente la différence de x et y.
- $-x$ représente l'opposé de x.
- $x \times y$ représente la multiplication de x par y.
- x / y lorsque que les opérandes sont de type entier, représente la division entière de x par y. Pour que la division entière ait un sens, il faut que y soit différent de 0. La division entière est définie de la manière suivante : soient $x \in \mathbb{N}$ et $y \in \mathbb{N}_1$, alors,
  $x / y = \text{max}(\{q \mid q \in \mathbb{N} \wedge y \times q \leq x\})$.
  On peut de façon équivalente donner les contraintes suivantes :
  si $q = x / y$ alors,
  $q \times y \leq x \wedge x < (q + 1) \times y \wedge q \geq 0$
  Puis cette définition est étendue aux entiers relatifs, grâce à la règle des signes (cf. *lois*).

  Lorsque les opérandes sont de type réel ou flottant alors cela représente l'inverse de la multiplication.
- x mod y représente le reste de la division entière de x par y. L'opérateur modulo n'est défini que pour les valeurs de x appartenant à $\mathbb{N}$ et pour des valeurs de y appartenant à $\mathbb{N}_1$.
- Si $x \in \mathbb{Z}$ et $y \in \mathbb{N}$, alors x^y représente x élevé à la puissance entière y.

  Sinon cela représente x élevé à l'exposant y.
- succ représente la fonction successeur, définie de $\mathbb{Z}$ dans $\mathbb{Z}$. succ (x) représente le successeur de x, c'est-à-dire $x + 1$.

- pred représente la fonction prédécesseur, définie de $\mathbb{Z}$ dans $\mathbb{Z}$. pred (x) représente le prédécesseur de x, c'est-à-dire $x - 1$.
- real représente la fonction de conversion de $\mathbb{Z}$ dans $\mathbb{R}$. real(x) est l'entier n tel que $n = x$

Soit x une expression de type $\mathbb{R}$ alors:

- floor représente la fonction partie entière, définie de $\mathbb{R}$ dans $\mathbb{Z}$. floor(x) représente la partie entière de x, i.e le seul entier n tel que $n \le x < n+1$.
- ceiling représente la fonction partie entière par excès, définie de $\mathbb{R}$ dans $\mathbb{Z}$. ceiling(x) représente la partie entière par excès de x, i.e. le seul entier n tel que $n\text{-}1 < x \le n$.

Lois

Soit $x \in \mathbb{N}$ alors $x^0 = 1$

Soient $x \in \mathbb{N}$ et $y \in \mathbb{N}\text{-}\{0\}$, alors :

$(-x) / y = - (x / y)$

$x / (-y) = - (x / y)$

$x \bmod y = x - y \times (x / y)$

Exemples

$b^2 - 4 \times \mathrm{a} \times \mathrm{c}$

$x - x^3 / 6 + x^5 / 120$

$x \times a + y^2 + z \bmod 9 - 7$

5.4 Expressions arithmétiques (suite)

Opérateur

max	Maximum
min	Minimum
card	Cardinal
Σ	Somme d'expressions arithmétiques
Π	Produit d'expressions arithmétiques

Syntaxe

Maximum	::=	"max" "(" *Expression* ")"
Minimum	::=	"min" "(" *Expression* ")"
Cardinal	::=	"card" "(" *Expression* ")"
Somme_généralisée	::=	"Σ" *Liste_ident* "." "(" *Prédicat* "|" *Expression* ")"
Produit_généralisé	::=	"Π" *Liste_ident* "." "(" *Prédicat* "|" *Expression* ")"

Règles de typage

Le type des expressions arithmétiques présentées ci-dessus est $\mathbb{Z}$ ou $\mathbb{R}$ (selon le type de l'opérande).

Dans les expressions : max (E), min (E), E doit être un ensemble de type $\mathbb{P}(\mathbb{Z})$ ou $\mathbb{P}(\mathbb{R})$.

Dans l'expression : card (E), E doit être un ensemble, de type $\mathbb{P}(T)$.

Dans les expressions : $\Sigma X . (P \mid E)$, $\Pi X . (P \mid E)$, les expressions E doivent être de type entier $\mathbb{Z}$.

Bonne définition

Expression	Condition de bonne définition
max (E)	E doit être non vide et doit posséder un majorant
min (E)	E doit être non vide et doit posséder un minorant
card (E)	E doit être fini
$\Sigma x . (P \mid E)$	l'ensemble $\{ x \mid P \}$ doit être fini
$\Pi x . (P \mid E)$	l'ensemble $\{ x \mid P \}$ doit être fini

Règle de portée

Dans les expressions : $\Sigma X . (P \mid E)$, $\Pi X . (P \mid E)$, la portée de la liste d'identificateurs X est le prédicat P et l'expression E.

Restriction

1. Les variables introduites par les expressions de la forme $\Sigma X . (P \mid E)$ ou $\Pi X . (P \mid E)$ doivent être typées par un prédicat de typage de données abstraites (cf. §3.3 *Typage des données abstraites*), situé dans une liste de conjonctions au plus haut niveau d'analyse syntaxique de P. Ces variables ne peuvent pas être utilisées dans P avant

d'avoir été typées.

Description

Soit E une expression représentant un ensemble non vide de nombres.

- $\mathsf{max}\ (E)$ représente le plus grand élément de E et $\mathsf{min}\ (E)$ représente le plus petit élément de E.

Soit F une expression qui représente un ensemble fini.

- $\mathsf{card}\ (F)$ représente le nombre d'éléments de F.

Soit X une liste de noms de variables deux à deux distincts. Soit P un prédicat qui type les variables de la liste X et E une expression de type entier ou réel.

- $\Sigma\ (X)\ .\ (P \mid E)$ représente la somme des expressions de E correspondant aux valeurs des variables X qui établissent P. Si $\{X \mid P\} = \varnothing$ alors la somme vaut 0.
- $\Pi(X)\ .\ (P \mid E)$ représente le produit des expressions de E correspondant aux valeurs des variables X qui établissent P. Si $\{X \mid P\} = \varnothing$ alors le produit vaut 1.

Exemples

Soit $E = \{-1, 2, 9, -6\}$,

$\mathsf{max}\ (E) = 9$ et $\mathsf{min}\ (E) = -6$

Soit $FRUITS = \{\ Fraise, Cassis, Framboise\ \}$,

$\mathsf{card}\ (\ FRUITS\ ) = 3$

$\Sigma\, x\ .\ (\ x \in \{\ 1, 2, 3\ \}\ \mid\ x + 1\ )\ =\ (\ 1 + 1\ ) + (\ 2 + 1\ ) + (\ 3 + 1\ )\ =\ 9$

$\Pi\, x\ .\ (\ x \in \mathbb{N}_1 \wedge x \leq 3\ \mid\ x\ )\ =\ 1 \times 2 \times 3\ =\ 6$

5.5 Expressions de couples

Opérateur

$\mapsto$ Correspondance binaire

Syntaxe

Couple	::=	*Expression* "$\mapsto$" *Expression*
	|	*Expression* "," *Expression*

Règle de typage

Si x et y sont respectivement de type T et U, alors $x \mapsto y$ est de type $T \times U$.

Description

Un couple est une paire ordonnée d'éléments, il se note $x \mapsto y$.

Une relation R d'un ensemble E dans un ensemble F est un ensemble de couples $(x \mapsto y)$ où x appartient à E et y appartient à F. Si $(i \mapsto j)$ est un élément d'une relation R, on dit que j est associé à i par R. Comme les relations sont des ensembles, tous les opérateurs sur les ensembles peuvent être appliqués à des relations.

Exemples

$$rel1 = \{(0 \mapsto \mathrm{FALSE}), (1 \mapsto \mathrm{TRUE}), (2 \mapsto \mathrm{FALSE}), (3 \mapsto \mathrm{TRUE}), (4 \mapsto \mathrm{FALSE}), (5 \mapsto \mathrm{TRUE})\}$$

$$rel2 = \{((0 \mapsto \mathrm{FALSE}) \mapsto 7), ((0 \mapsto \mathrm{TRUE}) \mapsto 9), ((1 \mapsto \mathrm{FALSE}) \mapsto 6), ((1 \mapsto \mathrm{TRUE}) \mapsto 8)\}$$

rel1 est une relation de $0 .. 5$ vers BOOL et *rel2* est une relation de $\{0, 1\} \times$ BOOL vers $6 .. 9$.

5.6 Ensembles prédéfinis

Opérateur

Ø	Ensemble vide
$\mathbb{Z}$	Ensemble des entiers relatifs
$\mathbb{N}$	Ensemble des entiers
$\mathbb{N}_1$	Ensemble des entiers non nuls
NAT	Ensemble des entiers concrets
NAT_1	Ensemble des entiers concrets non nuls
INT	Ensemble des entiers relatifs concrets
$\mathbb{R}$	Ensemble des nombres réels
FLOAT	Ensemble des nombres flottants
BOOL	Ensemble des booléens
STRING	Ensemble des chaînes de caractères

Syntaxe

Ensemble_vide	::=	"Ø"
Ensemble_nombres	::=	"$\mathbb{Z}$"
	|	"$\mathbb{N}$"
	|	"$\mathbb{N}_1$"
	|	"NAT"
	|	"NAT_1"
	|	"INT"
	|	"$\mathbb{R}$"
	|	"FLOAT"
Ensemble_booléen	::=	"BOOL"
Ensemble_chaînes	::=	"STRING"

Règles de typage

L'ensemble vide Ø n'a pas de type fixe établi. Il peut prendre le type de n'importe quel ensemble suivant le contexte où il se trouve. Son type est de la forme $\mathbb{P}(T)$.

Le type des ensembles $\mathbb{Z}$, $\mathbb{N}$, $\mathbb{N}_1$, NAT, NAT_1 et INT est $\mathbb{P}(\mathbb{Z})$.

Le type de l'ensembles $\mathbb{R}$ est $\mathbb{P}(\mathbb{R})$.

Le type de l'ensemble FLOAT est $\mathbb{P}$(FLOAT).

Le type de l'ensemble BOOL est $\mathbb{P}$(BOOL).

Le type de l'ensemble STRING est $\mathbb{P}$(STRING).

Restriction

Lors de chaque utilisation de l'ensemble vide Ø (dans un prédicat, une expression ou une substitution), le type de l'ensemble vide doit être instancié par le contexte.

Par exemple, le prédicat Ø = Ø est interdit. La substitution x := Ø est valide si la variable x est de type $\mathbb{P}(T)$, alors qu'elle est invalide si x n'a pas encore été typée.

Définitions

NAT $\triangleq$ 0 .. MAXINT

NAT_1 $\triangleq$ NAT - {0}

INT $\triangleq$ MININT .. MAXINT

BOOL $\triangleq$ {FALSE, TRUE}

Description

- L'ensemble vide $\varnothing$ est un ensemble qui ne possède pas d'élément. Il peut se définir comme la différence entre tout ensemble et lui-même, ce qui explique qu'il peut prendre le type de n'importe quel type ensemble.
- L'ensemble $\mathbb{Z}$ désigne l'ensemble des entiers relatifs. L'ensemble $\mathbb{N}$ désigne l'ensemble des entiers naturels. L'ensemble $\mathbb{N}_1$ désigne l'ensemble des entiers naturels strictement positifs.
- L'ensemble INT désigne l'ensemble des entiers relatifs concrets.
- L'ensemble NAT désigne l'ensemble des entiers naturels concrets.
- L'ensemble NAT_1 désigne l'ensemble des entiers naturels positifs concrets.
- L'ensemble $\mathbb{R}$ est celui des nombres réels.
- L'ensemble FLOAT est celui des nombres flottant.
- L'ensemble BOOL désigne l'ensemble des booléens.
- L'ensemble STRING désigne l'ensemble des chaînes de caractères.

5.7 Expressions ensemblistes

Opérateur

$\times$	Produit cartésien
$\mathbb{P}$	Ensemble des parties
$\mathbb{P}_1$	Ensemble des parties non vides
$\mathbb{F}$	Ensemble des parties finies
$\mathbb{F}_1$	Ensemble des parties finies non vides
{ | }	Ensemble défini en compréhension
{ }	Ensemble défini en extension
..	Intervalle

Syntaxe

Produit	::=	*Expression* "×" *Expression*
Ens_compréhension	::=	"{" Ident$^{+","}$ "|" *Prédicat* "}"
Sous_ensembles	::=	"$\mathbb{P}$" "(" *Expression* ")"
	|	"$\mathbb{P}_1$" "(" *Expression* ")"
Sous_ensembles_finis	::=	"$\mathbb{F}$" "(" *Expression* ")"
	|	"$\mathbb{F}_1$" "(" *Expression* ")"
Ens_extension	::=	"{" *Expression*$^{+","}$ "}"
Intervalle	::=	*Expression* ".." *Expression*

Définitions

$\mathbb{P}_1(E) \triangleq \{F \mid F \in \mathbb{P}(E) \land F \neq \varnothing\}$

$\mathbb{F}_1(E) \triangleq \{F \mid F \in \mathbb{F}(E) \land F \neq \varnothing\}$

Règles de typage

Soient X une expression de type ensemble $\mathbb{P}(T1)$ et Y une expression de type $\mathbb{P}(T2)$.

Le type de $X \times Y$ est $\mathbb{P}(T1 \times T2)$.

Le type de $\mathbb{P}(X)$, $\mathbb{P}_1(X)$, $\mathbb{F}(X)$ et $\mathbb{F}_1(X)$ est $\mathbb{P}(\mathbb{P}(T1))$.

Soient $E1$, ..., En des expressions de même type T, alors le type de $\{E1, ..., En\}$ est $\mathbb{P}(T)$.

Soient X une liste d'identificateurs deux à deux distincts $x1$, ..., xn typés dans le prédicat P et dont les types sont $T1$, ..., Tn. Alors, le type de l'ensemble en compréhension $\{X \mid P\}$ est $\mathbb{P}(T1 \times ... \times Tn)$. Dans le cas où X comporte un seul identificateur, le type de $\{X \mid P\}$ est $\mathbb{P}(T1)$.

Soient X et Y des expressions de type entier $\mathbb{Z}$, alors le type de $X .. Y$ est $\mathbb{P}(\mathbb{Z})$.

Règle de portée

Soient X une liste d'identificateurs et P un prédicat, alors dans l'ensemble en compréhension $\{X \mid P\}$, la portée des identificateurs de la liste X est le prédicat P.

Restrictions

1. Les variables X introduites par les expressions de la forme $\{ X \mid P \}$ doivent être deux à deux distinctes.
2. Les variables X introduites par les expressions de la forme $\{ X \mid P \}$ doivent être typées par un prédicat de typage de données abstraites (cf. §3.3 *Typage des données abstraites*), situé dans une liste de conjonctions au plus haut niveau d'analyse syntaxique de P. Ces variables ne peuvent pas être utilisées dans P avant d'avoir été typées.

Description

- Soient X et Y des ensembles, alors $X \times Y$ désigne le produit cartésien de X et Y, c'est-à-dire l'ensemble des couples dont le premier élément appartient à X et le second élément appartient à Y. Si X, Y et Z sont des ensembles, $X \times Y \times Z$ et $(X \times Y) \times Z$ désignent l'ensemble de couples de la forme $((x \mapsto y) \mapsto z)$, alors que $X \times (Y \times Z)$ désigne l'ensemble de couples de la forme $(x \mapsto (y \mapsto z))$.
- Soient $x1$, ..., xn des expressions, alors l'ensemble en extension $\{ x1, ..., xn\}$ représente l'ensemble dont les éléments sont $x1$, ..., xn.
- Soit E un ensemble, alors $\mathbb{P}(E)$ représente l'ensemble des parties de E. $\mathbb{P}_1(E)$ représente l'ensemble des parties non vides de E. $\mathbb{F}(E)$ représente l'ensemble des parties finies de E. $\mathbb{F}_1(E)$ représente l'ensemble des parties finies non vides de E.
- Soit X une liste d'identificateurs $x1$, ..., xn et P un prédicat qui type X et exprime des propriétés sur X. Alors, l'ensemble en compréhension $\{ X \mid P \}$ représente l'ensemble des maplets $(...(x1 \mapsto ...) \mapsto xn)$ qui vérifient P. Dans le cas où X comporte un seul identificateur, $\{ X \mid P \}$ représente l'ensemble des éléments $x1$ qui vérifient P.
- Soient x et y des entiers, alors l'intervalle $x .. y$ représente l'ensemble des entiers supérieurs ou égaux à x et inférieurs ou égaux à y. Ainsi dans le cas où $x > y$, $x .. y$ représente l'ensemble vide entier.

Exemples

Soient X et Y des ensembles définis en extension par : $X = \{ 1, 2, 3 \}$ et $Y = \{ 4, 5 \}$

$$X \times Y \ = \ \{ (1 \mapsto 4), (1 \mapsto 5), (2 \mapsto 4), (2 \mapsto 5), (3 \mapsto 4), (3 \mapsto 5) \}$$

$$\{ x \mid x \in X \land x \bmod 2 = 1 \} \ = \ \{ 1, 3 \}$$

$$\{ x, y \mid x \in \mathbb{N} \land y \in \mathbb{N} \land x < y \land y < 3\} \ = \ \{(0 \mapsto 1), (0 \mapsto 2), (1 \mapsto 2)\}$$

$$\mathbb{P}(X) \ = \ \{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

$$\mathbb{P}_1(X) \ = \ \{\{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

$$\mathbb{F}(X) = \mathbb{P}(X)$$

$$\mathbb{F}_1(X) = \mathbb{P}_1(X)$$

$$-1..5 \ = \ \{-1, 0, 1, 2, 3, 4, 5\}$$

$$6..4 = \varnothing$$

5.8 Expressions ensemblistes (suite)

Opérateur

-	Différence
$\cup$	Union
$\cap$	Intersection
union	Union généralisée
inter	Intersection généralisée
$\bigcup$	Union quantifiée
$\bigcap$	Intersection quantifiée

Syntaxe

Différence	::=	*Expression* "-" *Expression*
Union	::=	*Expression* "$\cup$" *Expression*
Intersection	::=	*Expression* "$\cap$" *Expression*
Union_généralisée	::=	"union" "(" *Expression* ")"
Intersection_généralisée	::=	"inter" "(" *Expression* ")"
Union_quantifiée	::=	"$\bigcup$" *Liste_ident* "." "(" *Prédicat* "|" *Expression* ")"
Intersection_quantifée	::=	"$\bigcap$" *Liste_ident* "." "(" Prédicat "|" *Expression* ")"

Définitions

Si $X \subseteq T$ et $Y \subseteq T$,

$$X - Y \ \hat{=}\ \{ x \mid x \in T \wedge (x \in X \wedge x \notin Y) \}$$

$$X \cup Y \ \hat{=}\ \{ x \mid x \in T \wedge (x \in X \vee x \in Y) \}$$

$$X \cap Y \ \hat{=}\ \{ x \mid x \in T \wedge (x \in X \wedge x \in Y) \}$$

Si $Z \in \mathbb{P}(\mathbb{P}(T))$,

$$\text{union}\,(Z) \ \hat{=}\ \{ x \mid x \in T \wedge \exists y . (y \in Z \wedge x \in y) \}$$

Si $Z \in \mathbb{P}_1(\mathbb{P}(T))$,

$$\text{inter}\,(Z) \ \hat{=}\ \{ x \mid x \in T \wedge \forall y . (y \in Z \Rightarrow x \in y) \}$$

Si $\forall x . (P \Rightarrow S \subseteq T))$,

$$\bigcup x . (P \mid S) \ \hat{=}\ \{ y \mid y \in T \wedge \exists z . (z \in T \wedge P \wedge y \in S) \}$$

Si $\forall x . (P \Rightarrow S \subseteq T))$ et $\exists x . (P)$,

$$\bigcap x . (P \mid S) \ \hat{=}\ \{ y \mid y \in T \wedge \forall z . (z \in T \wedge P \Rightarrow y \in S) \}$$

Règles de typage

Dans les expressions $X - Y$, $X \cup Y$ et $X \cap Y$, les ensembles X et Y doivent être du même type de la forme $\mathbb{P}(T)$. Le type de ces expressions est $\mathbb{P}(T)$.

Dans les expressions union (X) et inter (X), X doit être un ensemble d'ensembles, dont le type est $\mathbb{P}(\mathbb{P}(T))$. Le type de ces expressions est $\mathbb{P}(T)$.

Dans les expressions $\bigcup X . (P \mid S)$ et $\bigcap X . (P \mid S)$, X désigne une liste d'identificateurs, P est un prédicat qui doit typer X et S est un ensemble de type $\mathbb{P}(T)$. Le type de ces expressions est $\mathbb{P}(T)$.

Bonne définition

Expression	Condition de bonne définition
inter(E)	E doit être non vide
$\bigcap X . (P \mid E)$	$\{X \mid P\}$ doit être non vide

Règle de portée

Dans les expressions $\bigcup X . (P \mid S)$ et $\bigcap X . (P \mid S)$, la portée de la liste d'identificateurs X est le prédicat P et l'expression S.

Restriction

1. Les variables X introduites par les expressions de la forme $\bigcup X . (P \mid S)$ et $\bigcap X . (P \mid S)$ doivent être typées par un prédicat de typage de données abstraites (cf. §3.3 *Typage des données abstraites*), situé dans une liste de conjonctions au plus haut niveau d'analyse syntaxique de P. Ces variables ne peuvent pas être utilisées dans P avant d'avoir été typées.

Description

Soient E et F des ensembles.

- $E - F$, représente la différence des ensembles E et F, c'est-à-dire l'ensemble des éléments qui appartiennent à E mais pas à F.
- $E \cup F$ représente l'union des ensembles E et F, c'est-à-dire l'ensemble des éléments qui appartiennent à E ou à F.
- $E \cap F$ représente l'intersection des ensembles E et F c'est-à-dire l'ensemble des éléments qui appartiennent à E et à F.

Soit ENS un ensemble d'ensembles.

- union (ENS) représente l'union généralisée des éléments de ENS, c'est-à-dire l'ensemble obtenu par union des ensembles constituant les éléments de ENS.
- inter (ENS) représente l'intersection généralisée des éléments de ENS, c'est-à-dire l'ensemble obtenu par intersection des ensembles constituant les éléments de ENS.

Soient X une liste de variables, P un prédicat qui type la liste de variables X puis qui exprime une propriété sur X. Soit E un ensemble défini en fonction de X.

- $\bigcup X . (P \mid E)$ représente l'union des ensembles E indexés à l'aide d'une liste de variables X vérifiant le prédicat P. Si P est faux, alors l'union quantifiée représente l'ensemble vide.
- $\bigcap X . (P \mid E)$ représente l'intersection des ensembles E indexés à l'aide d'une liste de variables X vérifiant le prédicat P. Si P est faux, alors l'intersection quantifiée est dépourvue de sens.

Exemples

Soient $E = \{-1, 0, 3, 7, 8\}$ et $F = \{-3, -1, 4, 7, 9\}$,

$$E - F = \{0, 3, 8\}$$

$$E \cup F = \{-3, -1, 0, 3, 4, 7, 8, 9\}$$

$E \cap F \;=\; \{-1, 7\}$

Soit $S = \{\{1\}, \{1, 2\}, \{1, 3\}\}$,

union $(S) \;=\; \{1, 2, 3\}$

inter $(S) \;=\; \{1\}$

Soit $G \;=\; \{2, 4\}$,

$\bigcup y . (y \in G \mid \{z \mid z \in \mathbb{N} \wedge z \leq y\}) \;=\; \{0, 1, 2\} \cup \{0, 1, 2, 3, 4\} \;=\; \{0, 1, 2, 3, 4\}$

$\bigcap y . (y \in G \mid \{z \mid z \in \mathbb{N} \wedge z \leq y\}) \;=\; \{0, 1, 2\} \cap \{0, 1, 2, 3, 4\} \;=\; \{0, 1, 2\}$

5.9 Expressions de records

Opérateur

struct	Ensemble de records
rec	Record en extension
'	Accès à un champ de record (opérateur *quote*)

Syntaxe

Ensemble_records	::=	"struct" "(" (Ident ":" *Expression*)+"," ")"
Record_en_extension	::=	"rec" "(" ([Ident ":"] *Expression*)+"," ")"
Champ_de_record	::=	*Expression* "'" Ident

Règles de typage

Soit n un entier supérieur ou égal à 1 et i un entier compris entre 1 et n.

Dans l'expression struct (*Ident1* : *E1*, ..., *Identn* : *En*), *Ei* doit être de type $\mathbb{P}(Ti)$. Alors, le type de l'expression est $\mathbb{P}$(struct (*Ident1* : *T1*, ..., *Identn* : *Tn*)).

Dans l'expression rec (*Ident1* : *x1*, ..., *Identn* : *xn*), soit *Ti* le type de *xi*. Alors, le type de l'expression est struct (*Ident1* : *T1*, ..., *Identn* : *Tn*).

Dans l'expression rec (*x1*, ..., *xn*), soit *Ti* le type de chaque expression *xi*. Alors, le type de l'expression est de la forme struct (*Ident1* : *T1*, ..., *Identn* : *Tn*), où les *Identi* sont des identificateurs deux à deux distincts.

Dans l'expression *Record* ' *Identi*, *Record* doit être de type struct (*Ident1* : *T1*, ..., *Identn* : *Tn*), où *Identi* est le i$^{\text{ème}}$ label du type record. Alors, le type de l'expression est *Ti*.

Restrictions

1. Dans l'expression struct (*Ident1* : *E1*, ..., *Identn* : *En*), les noms de champs *Identi* doivent être deux à deux distincts.
2. Dans l'expression rec (*Ident1* : *x1*, ..., *Identn* : *xn*), les noms de champs *Identi* doivent être deux à deux distincts.
3. Un record en extension sans label, de la forme rec (*x1*, ..., *xn*), ne peut pas être utilisé pour typer une donnée.

Description

Soit n un entier supérieur ou égal à 1 et i un entier compris entre 1 et n.

- Soient *E1*, ..., *En* des ensembles et *Ident1*, ..., *Identn* des identificateurs deux à deux distincts, alors struct (*Ident1* : *E1*, ..., *Identn* : *En*) désigne un ensemble de données records. Cet ensemble est une collection ordonnée et non-vide des n ensembles *E1*, ..., *En* appelés champs de l'ensemble de records. Chaque champ possède un nom *Identi* appelé label.
- Soient *x1*, ..., *xn* des expressions et *Ident1*, ..., *Identn* des identificateurs deux à deux distincts, alors rec (*Ident1* : *x1*, ..., *Identn* : *xn*) désigne une donnée record, dont la valeur de chaque champ *Identi* est *xi*. Dans le cas où cette donnée record n'est pas utilisée pour typer une autre donnée (cf. *Typage des données abstraites*), alors les

labels sont facultatifs. L'écriture simplifiée rec (*x1*, ..., *xn*) peut être utilisée à la place de la précédente.

- Soit *rc* une donnée record dont l'un des labels est *identi*, alors l'expression *rc* ' *identi* construite à l'aide de l'opérateur *quote* désigne la valeur du champ *identi* de la donnée record *rc*.

Exemples

ENS_RES = struct (*Note* : 0 .. 20, *Suffisant* : BOOL) représente un ensemble de records à deux champs. Le premier champ s'appelle *Note* et désigne l'ensemble 0 .. 20. Le deuxième champ s'appelle *Suffisant* et désigne l'ensemble BOOL.

resultat = rec (*Note* : 12, *Suffisant* : TRUE) représente une donnée appartenant à l'ensemble *ENS_RES*. La valeur du champ *Note* est 12 et celle du champ *Suffisant* est TRUE. Si la donnée *resultat* a déjà été typée, alors l'écriture précédente peut être simplifiée en *resultat* = rec (12, TRUE).

resultat ' *Note* représente la valeur du champ *Note*, c'est-à-dire 12.

5.10 Ensembles de relations

Opérateur

$\leftrightarrow$ Ensemble des relations

Syntaxe

Ensemble_relations ::= *Expression* "$\leftrightarrow$" *Expression*

Définition

$X \leftrightarrow Y \ \hat{=}\ \mathbb{P}(X \times Y)$

Règle de typage

Dans l'expression $X \leftrightarrow Y$, X doit être de type $\mathbb{P}(T1)$ et Y doit être de type $\mathbb{P}(T2)$. Le type de $X \leftrightarrow Y$ est $\mathbb{P}(\mathbb{P}(T1 \times T2))$.

Description

Soient E et F des ensembles. Une relation de E dans F est un ensemble de couples $(x \mapsto y)$, où x est un élément de E et où y est un élément de F.

$E \leftrightarrow F$ désigne l'ensemble des relations de l'ensemble E dans l'ensemble F. C'est une autre écriture pour $\mathbb{P}(E \times F)$.

Exemples

$0..5 \leftrightarrow \mathrm{BOOL}$ représente l'ensemble des relations de l'intervalle $0..5$ dans l'ensemble BOOL. Les relations suivantes appartiennent à cet ensemble :

$rel1 = \{(0 \mapsto \mathrm{FALSE}), (1 \mapsto \mathrm{TRUE}), (2 \mapsto \mathrm{FALSE}), (3 \mapsto \mathrm{TRUE}), (4 \mapsto \mathrm{FALSE}), (5 \mapsto \mathrm{TRUE})\}$

$rel2 = \{(0 \mapsto \mathrm{FALSE}), (0 \mapsto \mathrm{TRUE}), (3 \mapsto \mathrm{TRUE})\}$

$rel3 = \varnothing$

5.11 Expressions de relations

Opérateur

id	Identité
r^{-1}	Inverse
prj_1	Première projection
prj_2	Deuxième projection
;	Composition
$\otimes$	Produit direct
$\|\|$	Produit parallèle

Syntaxe

Identité	::=	"id" "(" *Expression* ")"
Inverse	::=	*Expression* "-1"
Première_projection	::=	"prj₁" "(" *Expression* "," *Expression* ")"
Deuxième_projection	::=	"prj₂" "(" *Expression* "," *Expression* ")"
Composition	::=	*Expression* ";" *Expression*
Produit_direct	::=	*Expression* "⊗" *Expression*
Produit_parallèle	::=	*Expression* "||" *Expression*

Définitions

$$\text{id}\,(E) = \{\, x, y \mid x \in E \wedge y = x \,\}$$

Si $R \in X \leftrightarrow Y$,

$$R^{-1} \;\hat{=}\; \{\, y, x \mid (y \mapsto x) \in Y \leftrightarrow X \;\wedge\; (x \mapsto y) \in R \,\}$$

$$\text{prj}_1\,(E, F) \;\hat{=}\; \{\, x, y, \mathrm{z} \mid x, y, \mathrm{z} \in E \times F \times E \;\wedge\; \mathrm{z} = x \,\}$$

$$\text{prj}_2\,(E, F) \;\hat{=}\; \{\, x, y, \mathrm{z} \mid x, y, \mathrm{z} \in E \times F \times F \;\wedge\; \mathrm{z} = y \,\}$$

Si $R1 \in T \leftrightarrow U$ et $R2 \in U \leftrightarrow V$,

$$R1 \,;\, R2 \;\hat{=}\; \{\, x, \mathrm{z} \mid x, z \in T \times V \wedge \exists\, y\,.\,(\, y \in U \wedge (x \mapsto y) \in R1 \wedge (y \mapsto \mathrm{z}) \in R2 \,)\,\}$$

Si $R1 \in T \leftrightarrow U$ et $R2 \in T \leftrightarrow V$,

$$R1 \otimes R2 \;\hat{=}\; \{\, x, (y, z) \mid x, (y, \mathrm{z}) \in T \times (U \times V) \wedge (x \mapsto y) \in R1 \wedge (x \mapsto \mathrm{z}) \in R2 \,\}$$

Si $R1 \in T \leftrightarrow U$ et $R2 \in V \leftrightarrow W$,

$$R1 \,||\, R2 \;\hat{=}\; \{\, (x, y), (z, a) \mid (x, y), (z, a) \in (T \times V) \times (U \times W) \wedge (x \mapsto z) \in R1 \wedge (y \mapsto a) \in R2 \,\}$$

Règles de typage

Dans l'expression $\text{id}\,(E)$, E doit être de type $\mathbb{P}(T)$. Alors $\text{id}\,(E)$ est de type $\mathbb{P}(T \times T)$.

Dans l'expression R^{-1}, R doit être de type $\mathbb{P}(T \times U)$. Alors R^{-1} est une relation de type $\mathbb{P}(U \times T)$.

Dans les expressions $\text{prj}_1\,(E, F)$ et $\text{prj}_2\,(E, F)$, E et F doivent être de type $\mathbb{P}(T)$ et $\mathbb{P}(U)$. Alors $\text{prj}_1\,(E, F)$ est une relation de type $\mathbb{P}(T \times U \times T)$ et $\text{prj}_2\,(E, F)$ est une relation de type $\mathbb{P}(T \times U \times U)$.

Dans l'expression $(\, E \,;\, F \,)$, E doit être de type $\mathbb{P}(T \times U)$ et F doit être de type $\mathbb{P}(U \times V)$. Alors $E \,;\, F$ est une relation de type $\mathbb{P}(T \times V)$.

Dans l'expression $E \otimes F$, E doit être de type $\mathbb{P}(T\times U)$ et F doit être de type $\mathbb{P}(T\times V)$. Alors $E \otimes F$ est une relation de type $\mathbb{P}(T\times(U\times V))$.

Dans l'expression $E \parallel F$, E doit être de type $\mathbb{P}(T\times U)$ et F doit être de type $\mathbb{P}(V\times W)$. Alors, $E \parallel F$ est une relation de type $\mathbb{P}((T\times V)\times(U\times W))$.

Restriction

Les opérateurs ; et || lorsqu'ils représentent la composition de deux relations et le produit parallèle de deux relations ne doivent pas apparaître s'il peut y avoir ambiguïté avec les opérateurs désignant des substitutions en séquence ou simultanée. Pour lever l'ambiguïté, il est toujours possible de parenthéser l'expression. Par exemple, $R3 := R1 ; R2$ est interdit car ambigu. À la place, il faut écrire $R3 := (R1 ; R2)$.

Description

- Soit E un ensemble, id (E) représente la relation identité construite sur E, c'est-à-dire la relation qui à tout élément de E associe ce même élément.
- Soit R une relation, R^{-1} représente la relation inverse de R, c'est-à-dire la relation composée des couples inverses de ceux de R. Si $(x \mapsto y) \in R$ alors $(y \mapsto x) \in R^{-1}$.

Soient X et Y des ensembles,

- $\mathrm{prj}_1(X, Y)$ représente la relation première projection de $X \times Y$ dans X, qui à tout couple $(x \mapsto y)$ de $X \times Y$ associe le premier composant x du couple.
- $\mathrm{prj}_2(X, Y)$ représente la relation deuxième projection de $X \times Y$ dans Y, qui à tout couple $(x \mapsto y)$ de $X \times Y$ associe le deuxième composant y du couple.
- Soient $R1$ une relation de l'ensemble A vers l'ensemble B et $R2$ une relation de l'ensemble B vers l'ensemble C. Alors $(R1 ; R2)$ représente la composition de $R1$ et $R2$. Elle contient l'ensemble des couples $(a \mapsto c)$ tels qu'il existe un élément b de B tel que $(a \mapsto b) \in R1$ et $(b \mapsto c) \in R2$.
- Soient $R1$ une relation de l'ensemble A vers l'ensemble B et $R2$ une relation de l'ensemble A vers l'ensemble C. Alors $R1 \otimes R2$ représente le produit direct de $R1$ et $R2$. Cette relation contient l'ensemble des couples $a \mapsto (b \mapsto c)$ tels qu'il existe un couple $(a \mapsto b)$ de $R1$ et un couple $(a \mapsto c)$ de $R2$.
- Soient $R1$ une relation de l'ensemble A vers l'ensemble B et $R2$ une relation de l'ensemble C vers l'ensemble D. Alors $(R1 \parallel R2)$ représente le produit parallèle de $R1$ et $R2$. Cette relation contient l'ensemble des couples de la forme $((a \mapsto c) \mapsto (b \mapsto d))$ tels qu'il existe un couple $(a \mapsto b)$ de $R1$ et un couple $(c \mapsto d)$ de $R2$.

Exemples

Soit $E = \{3, 5\}$,

$\mathrm{id}\ (E) = \{(3 \mapsto 3), (5 \mapsto 5)\}$

Soit $R1 = \{(0 \mapsto 4), (2 \mapsto 4), (2 \mapsto 7), (3 \mapsto 3)\}$,

$R1^{-1} = \{(4 \mapsto 0), (4 \mapsto 2), (7 \mapsto 2), (3 \mapsto 3)\}$

Soient $E = \{0, 1\}$ et $F = \{-1, 2\}$,

$\mathrm{prj}_1\ (E, F) = \{((0 \mapsto -1) \mapsto 0), ((0 \mapsto 2) \mapsto 0), ((1 \mapsto -1) \mapsto 1), ((1 \mapsto 2) \mapsto 1)\}$

$\mathrm{prj}_2\ (E, F) = \{((0 \mapsto -1) \mapsto -1), ((0 \mapsto 2) \mapsto 2), ((1 \mapsto -1) \mapsto -1), ((1 \mapsto 2) \mapsto 2)\}$

Soient $R1 = \{(0 \mapsto 2), (1 \mapsto 5), (2 \mapsto 5), (3 \mapsto 7)\}$ et $R2 = \{(0 \mapsto 0), (2 \mapsto -1), (5 \mapsto 8), (6 \mapsto 9)\}$,

$(R1 \; ; R2) = \{(0 \mapsto -1), (1 \mapsto 8), (2 \mapsto 8)\}$

Soient $R1 = \{(0 \mapsto 0), (1 \mapsto 10), (2 \mapsto 20)\}$ et $R2 = \{(0 \mapsto 0), (1 \mapsto 20), (2 \mapsto 40), (3 \mapsto 60)\}$,

$$R1 \otimes R2 = \{ (0 \mapsto (0 \mapsto 0)), \\ (1 \mapsto (10 \mapsto 20)), \\ (2 \mapsto (20 \mapsto 40))\}$$

Soient $R1 = \{(0 \mapsto 7), (1 \mapsto 6)\}$ et $R2 = \{(10 \mapsto 11), (12 \mapsto 12)\}$,

$$(R1 \parallel R2) = \{ ((0 \mapsto 10) \mapsto (7 \mapsto 11)), \\ ((0 \mapsto 12) \mapsto (7 \mapsto 12)), \\ ((1 \mapsto 10) \mapsto (6 \mapsto 11)), \\ ((1 \mapsto 12) \mapsto (6 \mapsto 12))\}$$

5.12 Expressions de relations (suite)

Opérateur

R^{n}	Itération
R^{*}	Fermeture transitive et réflexive
R^{+}	Fermeture transitive

Syntaxe

Itération	::=	*Expression* $^{\mathit{Expression}}$
Fermeture_réflexive	::=	*Expression* $^{"*"}$
Fermeture	::=	*Expression* $^{"+"}$

Définitions

Soit R une relation d'un ensemble E dans lui-même et soit n un entier naturel.

$$R^0 \ \hat{=}\ \mathrm{id}(\mathrm{dom}(R))$$
$$R^1 \ \hat{=}\ R$$
$$R^{n+1} \ \hat{=}\ R\ ;\ R^n$$
$$R^* \ \hat{=}\ \bigcup n\,.\,(n \in \mathbb{N} \mid R^{\mathrm{n}})$$
$$R^+ \ \hat{=}\ \bigcup n\,.\,(n \in \mathbb{N}_1 \mid R^{\mathrm{n}})$$

Notons que la définition de R^0 s'éloigne légèrement de celle du *B-Book* (mais est plus utilisable en pratique).

Règles de typage

Dans l'expression R^n, R est de type $\mathbb{P}(T\times T)$ et n est de type $\mathbb{Z}$. Le type de l'expression est $\mathbb{P}(T\times T)$.

Dans les expressions $R1^*$ et $R1^+$, $R1$ doit être de type $\mathbb{P}(T\times T)$. Le type des expressions est $\mathbb{P}(T\times T)$.

Bonne définition

Expression	Condition de bonne définition
R^n	n doit appartenir à $\mathbb{N}$

Description

Soit R une relation d'un ensemble E dans lui-même et soit n un entier naturel.

- R^n représente la relation R itérée n fois par rapport à l'opérateur de composition.
- R^* représente la fermeture transitive et réflexive de R. C'est la plus petite relation contenant R qui soit transitive et réflexive.
- R^+ représente la fermeture transitive de R. C'est la plus petite relation contenant R qui soit transitive.

Exemples

Soit $E = \{1, 2, 3\}$, $R = \{(1 \mapsto 3), (2 \mapsto 1), (2 \mapsto 2), (3 \mapsto 3)\}$,

$$R^1 = R$$

$$R^2 = \{(1 \mapsto 3), (2 \mapsto 1), (2 \mapsto 2), (2 \mapsto 3), (3 \mapsto 3)\}$$
$$R^+ = R^2$$
$$R^* = \{(1 \mapsto 1), (1 \mapsto 3), (2 \mapsto 1), (2 \mapsto 2), (2 \mapsto 3), (3 \mapsto 3)\}$$

5.13 Expressions de relations (suite)

Opérateur

dom	Domaine
ran	Codomaine
[ ]	Image

Syntaxe

Domaine	::=	"dom" "(" *Expression* ")"
Codomaine	::=	"ran" "(" *Expression* ")"
Image	::=	*Expression* "[" *Expression* "]"

Définitions

Si $R \in X \leftrightarrow Y$,

$$\text{dom}\,(R) \triangleq \{ x \mid x \in X \land \exists y \,.\, (y \in Y \land (x \mapsto y) \in R) \}$$

$$\text{ran}\,(R) \triangleq \{ y \mid y \in Y \land \exists x \,.\, (x \in X \land (x \mapsto y) \in R) \}$$

Si $R \in X \leftrightarrow Y$ et $F \subseteq X$,

$$R\,[F] \triangleq \{ y \mid y \in Y \land \exists x \,.\, (x \in F \land (x \mapsto y) \in R) \}$$

Règles de typage

Dans les expressions dom (R) et ran (R), R doit être une relation de type $\mathbb{P}(T \times V)$. Alors le type de dom (R) est $\mathbb{P}(T)$ et le type de ran (R) est $\mathbb{P}(V)$.

Dans l'expression $R\,[E]$, R doit être une relation de type $\mathbb{P}(T \times V)$ et E doit être un ensemble de type $\mathbb{P}(T)$. Alors l'expression est de type $\mathbb{P}(V)$.

Description

Soit R une relation d'un ensemble A vers un ensemble B.

- dom (R) désigne le domaine de R, c'est-à-dire l'ensemble des éléments a de A pour lesquels il existe un élément b de B tel que $(a \mapsto b) \in R$.
- ran (R) désigne le codomaine de R (*range* en anglais), c'est-à-dire l'ensemble des éléments b de B pour lesquels il existe un élément a de A tel que $(a \mapsto b) \in R$.

Soit E une partie de A,

- $R\,[E]$ désigne l'image de E par R. C'est l'ensemble des éléments de B qui sont associés à un élément de E par la relation R.

Exemples

Soit $R = \{(0 \mapsto 4), (2 \mapsto 4), (2 \mapsto 7), (3 \mapsto 3)\}$,

dom $(R) = \{0, 2, 3\}$

ran $(R) = \{4, 7, 3\}$

Soit $E = \{-1, 0, 1, 2\}$,

$R\,[E] = \{4, 7\}$

5.14 Expressions de relations (suite)

Opérateur

$\lhd$	Restriction sur le domaine
$⩤$	Soustraction sur le domaine
$\rhd$	Restriction sur le codomaine
$⩥$	Soustraction sur le codomaine
$\Leftarrow\!\!+$	Surcharge

Syntaxe

Restriction_domaine	::=	*Expression* "$\lhd$" *Expression*
Soustraction_domaine	::=	*Expression* "$⩤$" *Expression*
Restriction_codomaine	::=	*Expression* "$\rhd$" *Expression*
Soustraction_codomaine	::=	*Expression* "$⩥$" *Expression*
Surcharge	::=	*Expression* "$\Leftarrow\!\!+$" *Expression*

Définitions

Si $R \in X \leftrightarrow Y$ et $F \subseteq X$,

$$F \lhd R \;=\; \{\, x, y \mid (x \mapsto y) \in R \;\wedge\; x \in F \,\}$$

$$F ⩤ R \;=\; \{\, x, y \mid (x \mapsto y) \in R \;\wedge\; x \notin F \,\}$$

Si $R \in X \leftrightarrow Y$ et $F \subseteq Y$,

$$R \rhd F = \{\, x, y \mid (x \mapsto y) \in R \wedge y \in F \,\}$$

$$R ⩥ F = \{\, x, y \mid (x \mapsto y) \in R \wedge y \notin F \,\}$$

Si $R \in X \leftrightarrow Y$ et $Q \in X \leftrightarrow Y$,

$$Q \Leftarrow\!\!+ R = \{\, x, y \mid (x, y) \in X \times Y \wedge (\, ((x \mapsto y) \in Q \wedge x \notin \mathrm{dom}\,(R)) \vee (x \mapsto y) \in R \,)\,\}$$

Règles de typage

Dans les expressions $X \lhd R$ et $X ⩤ R$, R doit être une relation de type $\mathbb{P}(T \times V)$ et X doit être un ensemble de type $\mathbb{P}(T)$. Le type des expressions est $\mathbb{P}(T \times V)$.

Dans les expressions $R \rhd Y$ et $R ⩥ Y$, R doit être une relation de type $\mathbb{P}(T \times V)$ et Y doit être un ensemble de type $\mathbb{P}(V)$. Le type des expressions est $\mathbb{P}(T \times V)$.

Dans l'expression $R1 \Leftarrow\!\!+ R2$, $R1$ et $R2$ doivent être des relations de type $\mathbb{P}(T \times V)$. Le type de l'expression est $\mathbb{P}(T \times V)$.

Description

Soient R, $R1$ et $R2$ des relations, E et F des ensembles.

- $E \lhd R$ désigne la restriction sur le domaine de R à l'ensemble E. C'est l'ensemble des couples $(x \mapsto y)$ de R pour lesquels x appartient à E.
- $E ⩤ R$ désigne la soustraction sur le domaine de R à l'ensemble E. C'est l'ensemble des couples $(x \mapsto y)$ de R pour lesquels x n'appartient pas à E.
- $R \rhd F$ désigne la restriction sur le codomaine de R à l'ensemble F. C'est l'ensemble des couples $(x \mapsto y)$ de R pour lesquels y appartient à F.

- $R ⩥ F$ désigne la soustraction sur le codomaine de R à l’ensemble F. C’est l’ensemble des couples $(x \mapsto y)$ de R pour lesquels y n’appartient pas à F.
- $R1 \mathbin{\lhd\mkern-14mu+} R2$ désigne la surcharge de $R1$ par $R2$. C’est la relation constituée des éléments de $R2$ et des éléments de $R1$ dont le premier élément n’appartient pas au domaine de $R2$. Ainsi dans la relation obtenue, les éléments de $R2$ notés $(x \mapsto z)$ surchargent les éventuels éléments $(x \mapsto y)$ de $R1$.

Exemples

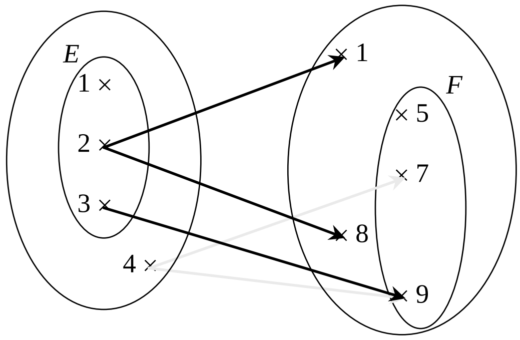
Soit la relation $R = \{(2 \mapsto 1), (2 \mapsto 8), (3 \mapsto 9), (4 \mapsto 7), (4 \mapsto 9)\}$,

soient les ensembles $E = \{1, 2, 3\}$ et $F = \{5, 7, 9\}$,

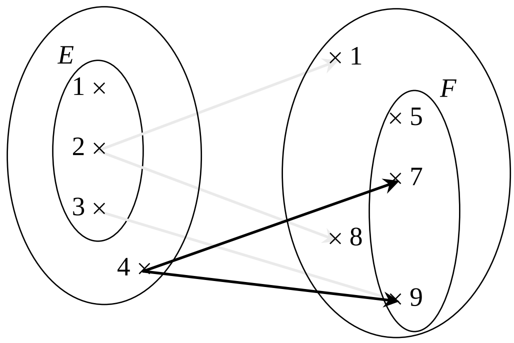
$E \lhd R = \{(2 \mapsto 1), (2 \mapsto 8), (3 \mapsto 9)\}$

$E ⩤ R = \{(4 \mapsto 7), (4 \mapsto 9)\}$

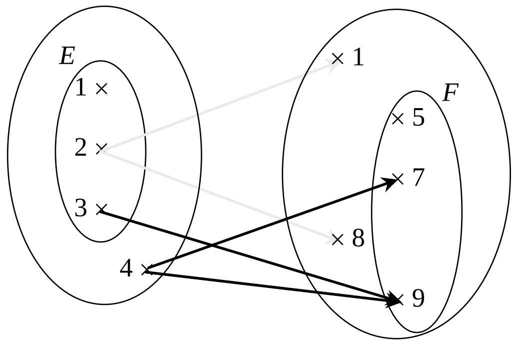
$R \rhd F =\{(3 \mapsto 9), (4 \mapsto 7), (4 \mapsto 9)\}$



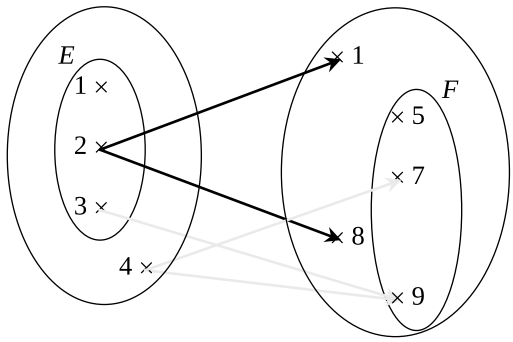
$E \lhd R$



$E ⩤ R$



$R \rhd F$



$R ⩥ F$

$R ⩥ F = \{(2 \mapsto 1), (2 \mapsto 8)\}$

Soient les relations $R1 = \{(2 \mapsto 1), (2 \mapsto 8), (3 \mapsto 9), (4 \mapsto 7), (4 \mapsto 9)\}$ et $R2 = \{(0 \mapsto -1), (1 \mapsto 7), (2 \mapsto 9)\}$,

$R1 \mathbin{\lhd\mkern-14mu+} R2 = \{(0 \mapsto -1), (1 \mapsto 7), (2 \mapsto 9), (3 \mapsto 9), (4 \mapsto 7), (4 \mapsto 9)\}$

5.15 Ensembles de fonctions

Opérateur

$\pfun$	Fonctions partielles
$\rightarrow$	Fonctions totales
$\pinj$	Injections partielles
$\rightarrowtail$	Injections totales
$\psurj$	Surjections partielles
$\twoheadrightarrow$	Surjections totales
$\rightarrowtail\!\!\!\!\twoheadrightarrow$	Bijections totales

Syntaxe

Fonction_partielle	::=	*Expression* "$\pfun$" *Expression*
Fonction_totale	::=	*Expression* "$\rightarrow$" *Expression*
Injection_partielle	::=	*Expression* "$\pinj$" *Expression*
Injection_totale	::=	*Expression* "$\rightarrowtail$" *Expression*
Surjection_partielle	::=	*Expression* "$\psurj$" *Expression*
Surjection_totale	::=	*Expression* "$\twoheadrightarrow$" *Expression*
Bijection_totale	::=	*Expression* "$\rightarrowtail\!\!\!\!\twoheadrightarrow$" *Expression*

Définitions

$X \pfun Y \;\hat{=}\; \{ r \mid r \in X \leftrightarrow Y \wedge (r^{-1} ; r) \subseteq \mathrm{id}(Y) \}$

$X \rightarrow Y \;\hat{=}\; \{ f \mid f \in X \pfun Y \wedge \mathrm{dom}(f) = X \}$

$X \pinj Y \;\hat{=}\; \{ f \mid f \in X \pfun Y \wedge f^{-1} \in Y \pfun X \}$

$X \rightarrowtail Y \;\hat{=}\; X \pinj Y \cap X \rightarrow Y$

$X \psurj Y \;\hat{=}\; \{ f \mid f \in X \pfun Y \wedge \mathrm{ran}(f) = Y \}$

$X \twoheadrightarrow Y \;\hat{=}\; X \psurj Y \cap X \rightarrow Y$

$X \rightarrowtail\!\!\!\!\twoheadrightarrow Y \;\hat{=}\; X \rightarrowtail Y \cap X \twoheadrightarrow Y$

Règle de typage

Dans les expressions $X \pfun Y$, $X \rightarrow Y$, $X \pinj Y$, $X \rightarrowtail Y$, $X \psurj Y$, $X \twoheadrightarrow Y$, $X \rightarrowtail\!\!\!\!\twoheadrightarrow Y$, les expressions X et Y sont de types $\mathbb{P}(T1)$ et $\mathbb{P}(T2)$. Le type des expressions est $\mathbb{P}(\mathbb{P}(T1 \times T2))$.

Description

Soient X et Y des ensembles.

- $X \pfun Y$ désigne l'ensemble des fonctions partielles de X dans Y. Une fonction partielle de X dans Y est une relation qui ne contient pas deux couples distincts ayant le même premier élément.
- $X \rightarrow Y$ désigne l'ensemble des fonctions totales de X dans Y. Une fonction totale de X dans Y est une fonction partielle dont le domaine est exactement X (et n'est pas seulement inclus dans X comme c'est le cas pour une fonction partielle).
- $X \pinj Y$ désigne l'ensemble des injections partielles de X dans Y. Une injection partielle de X dans Y est une fonction partielle qui à deux éléments distincts de X

associe deux éléments distincts de Y. L'inverse d'une injection partielle de X dans Y est donc une fonction partielle de Y dans X. Le concept d'injection totale, dont le symbole est '$\rightarrowtail$', se définit de manière similaire.

- $X \twoheadrightarrow\!\!\!\!\!+\; Y$ désigne l'ensemble des surjections partielles de X dans Y. Une surjection partielle de X dans Y est une fonction partielle qui est telle que chaque élément de Y est en correspondance avec un élément de X au moins. Le concept de surjection totale, dont le symbole est '$\twoheadrightarrow$', se définit de manière similaire.

Exemples

Si $r1 = \{(0 \mapsto 1), (1 \mapsto 2), (2 \mapsto 2)\}$,
alors $r1 \in \{0, 1, 2, 3\} \nrightarrow \{0, 1, 2\}$
et $r1 \in \{0, 1, 2\} \rightarrow \{0, 1, 2\}$

Si $r2 = \{(0 \mapsto 1), (1 \mapsto 2), (2 \mapsto 3)\}$,
alors $r2 \in \{0, 1, 2, 3\} \rightarrowtail\!\!\!\!\!+\; \{0, 1, 2, 3\}$
et $r2 \in \{0, 1, 2\} \rightarrowtail \{0, 1, 2, 3\}$

Si $r3 = \{(0 \mapsto 1), (1 \mapsto 2), (2 \mapsto 2)\}$,
alors $r3 \in \{0, 1, 2, 3\} \twoheadrightarrow\!\!\!\!\!+\; \{1, 2\}$
et $r3 \in \{0, 1, 2\} \twoheadrightarrow \{1, 2\}$

Si $r4 = \{(0 \mapsto 1), (1 \mapsto 2), (2 \mapsto 3)\}$,
alors $r4 \in \{0, 1, 2\} \rightarrowtail\!\!\!\!\!\twoheadrightarrow \{1, 2, 3\}$

5.16 Expressions de fonctions

Opérateur

λ	Lambda expression
$f\,(\,)$	Évaluation de fonction
fnc	Transformée en fonction
rel	Transformée en relation

Syntaxe

Lambda_expression	::=	"λ" *Liste_ident* "." "(" *Prédicat* "|" *Expression* ")"
Évaluation_fonction	::=	*Expression* "(" *Expression* ")"
Transformée_fonction	::=	"fnc" "(" *Expression* ")"
Transformée_relation	::=	"rel" "(" *Expression* ")"

Définitions

Si $\forall x \,.\, (x \in T \Rightarrow E \in U)$,

$\lambda\, x \,.\, (x \in T \wedge P \mid E) \;\hat{=}\; \{\, x, y \mid x, y \in T \times U \wedge P \wedge y = E \,\}$ où y n'est pas libre dans x, T, P et E

Si $f \in T \mathrel{+\!\!\!\rightarrow} U$ et $E \in \mathrm{dom}\,(f)$,

$f(E) \;\hat{=}\; \mathrm{choice}\;(\,f\,[\{E\}]\,)$

On rappelle que l'opérateur choice (cf. [B-Book] §2.1.2) appliqué à un ensemble non vide, désigne un élément “privilégié” de cet ensemble. Dans le cas qui nous occupe ici, l'ensemble en question, $f\,[\{E\}]$, n'a qu'un seul élément. L'élément privilégié de cet ensemble ne peut donc être que cet élément-là. Attention, cet opérateur choice ne doit pas être confondu avec la substitution « choix borné » (cf. §6.6) qui utilise le mot-clé CHOICE.

Soit R une relation de X vers Y,

$\mathrm{fnc}\;(R) \;\hat{=}\; \lambda\, x \,.\, (x \in \mathrm{dom}\;(R) \mid R\;[\{x\}])$

Soit F une fonction de X vers $\mathbb{P}(Y)$,

$\mathrm{rel}\;(F) \;\hat{=}\; \{\, x, y \mid x, y \in \mathrm{dom}\;(F) \times Y \;\wedge\; y \in F(x))$

Règles de typage

Dans l'expression $\lambda X.(P \mid E)$, X désigne une liste d'identificateurs deux à deux distincts, P est un prédicat qui doit commencer par typer tous les éléments de X et E est une expression de type T. Alors l'expression est de type $\mathbb{P}(T1 \times ... \times Tn \times T)$.

Dans l'expression $f\,(y)$, f est une fonction de type $\mathbb{P}(T1 \times T2)$ et y doit être de type $T1$. Le type de l'expression est $T2$.

Dans l'expression fnc (R), R doit être une relation de type $\mathbb{P}(T1 \times T2)$. Le type de l'expression est $\mathbb{P}(T1 \times \mathbb{P}(T2))$.

Dans l'expression rel (R), R représente une relation dont le type doit être de la forme $\mathbb{P}(T1 \times \mathbb{P}(T2))$. Le type de l'expression est $\mathbb{P}(T1 \times T2)$.

Bonne définition

Expression	Condition de bonne définition
$f(x)$	$x \in \mathrm{dom}(f) \wedge f \in \mathrm{dom}(f) \rightarrow \mathrm{ran}(f)$
$rel(f)$	$f \in \mathrm{dom}(f) \rightarrow \mathrm{ran}(f)$

Règle de portée

Dans l'expression $\lambda X . (P \mid E)$, la portée des identificateurs X est le prédicat P et l'expression E.

Restriction

1. Les variables X introduites par les expressions de la forme $\lambda X . (P \mid E)$ doivent être typées par un prédicat de typage de données abstraites (cf. §3.3 *Typage des données abstraites*), situé dans une liste de conjonctions au plus haut niveau d'analyse syntaxique de P. Ces variables ne peuvent pas être utilisées dans P avant d'avoir été typées.

Description

- Une lambda expression permet de définir une fonction par la valeur qu'elle prend en chaque point de son domaine. Soient x un identificateur, P un prédicat qui commence par typer x et E une expression qui dépend de x. Alors $\lambda x . (P \mid E)$ désigne une lambda expression. C'est la fonction constituée des couples $(x \mapsto E)$ pour chaque élément x vérifiant P.
- Soit f une fonction de X dans Y et soit x un élément de X. Alors $f(x)$ désigne l'unique élément y de Y tel que le couple $(x \mapsto y)$ appartienne à f. Pour que l'expression ait un sens, il faut que x appartienne au domaine de f.
- Soit R une relation de X dans Y. Alors $\mathrm{fnc}(R)$ désigne la transformée en fonction de relation R. C'est la fonction de X dans $\mathbb{P}(Y)$ qui à chaque élément x du domaine de R associe l'ensemble des éléments de Y liés à x par la relation R.
- Soit Fct une fonction de X dans $\mathbb{P}(Y)$. Alors $\mathrm{rel}(Fct)$ désigne la transformée en relation de Fct. C'est la relation de X dans Y constituée des couples $(x \mapsto y)$ tels que x appartienne au domaine de Fct et que y appartienne à l'élément associé à x par la fonction Fct.

Exemples

La lambda expression : $\lambda x . (x \in \mathbb{Z} \mid x \times 2)$ définit la fonction multiplication par 2 sur $\mathbb{Z}$.

Soit la fonction $f = \{(0 \mapsto 6), (1 \mapsto 2), (3 \mapsto 6), (4 \mapsto -5)\}$,

$f(3) = 6$

Soit la relation $R = \{(0 \mapsto 1), (0 \mapsto 2), (1 \mapsto 1), (1 \mapsto 7), (2 \mapsto 3)\}$,

$\mathrm{fnc}(R) = \{(0 \mapsto \{1, 2\}), (1 \mapsto \{1, 7\}), (2 \mapsto \{3\})\}$

Soit la fonction $f = \{(-1 \mapsto \{0, 2\}), (1 \mapsto \{6, 8\}), (3 \mapsto \{3\})\}$,

$\mathrm{rel}(f) = \{(-1 \mapsto 0), (-1 \mapsto 2), (1 \mapsto 6), (1 \mapsto 8), (3 \mapsto 3)\}$

5.17 Ensembles de suites

Opérateur

seq	Suites
seq_1	Suites non vides
iseq	Suites injectives
iseq_1	Suites injectives non vides
perm	Permutations
[]	Suite vide
[ ]	Suite en extension

Syntaxe

Suites	::=	"seq" "(" *Expression* ")"
Suites_non_vide	::=	"seq_1" "(" *Expression* ")"
Suites_injectives	::=	"iseq" "(" *Expression* ")"
Suites_inj_non_vide	::=	"iseq_1" "(" *Expression* ")"
Permutations	::=	"perm" "(" *Expression* ")"
Suite_vide	::=	"[]"
Suite_extension	::=	"[" $Expression^{+","}$ "]"

Règles de typage

Dans les expressions $\text{seq}\,(E)$, $\text{seq}_1\,(E)$, $\text{iseq}\,(E)$, $\text{iseq}_1\,(E)$, $\text{perm}\,(E)$, E doit désigner un ensemble de type $\mathbb{P}(T)$. Alors les expressions sont de type $\mathbb{P}(\mathbb{P}(\mathbb{Z}\times T))$.

La suite vide [] n'a pas de type fixe établi. Elle peut prendre le type de n'importe quelle suite. Son type doit être de la forme $\mathbb{P}(\mathbb{Z}\times T)$.

Dans la suite en extension $[E1, ..., En]$, les éléments $E1, ..., En$ de la suite doivent tous être du même type T. Le type de la suite est alors $\mathbb{P}(\mathbb{Z}\times T)$.

Bonne définition

Expression	Condition de bonne définition
perm (E)	E doit être un ensemble fini

Définitions

$\text{seq}\,(E) \;\hat{=}\; \bigcup n \,.\, (n \in \mathbb{N} \mid 1..n \rightarrow E)$

$\text{seq}_1\,(E) \;\hat{=}\; \text{seq}\,(E) - \{\varnothing\}$

$\text{iseq}\,(E) \;\hat{=}\; \{s \mid s \in \text{seq}\,(E) \wedge s \in \mathbb{N}_1 \rightarrowtail\!\!\!\!\!+ \; E\}$

$\text{iseq}_1\,(E) \;\hat{=}\; \text{iseq}\,(E) - \{\varnothing\}$

$\text{perm}\,(E) \;\hat{=}\; \{s \mid s \in \text{iseq}\,(E) \wedge s \in \mathbb{N}_1 \twoheadrightarrow\!\!\!\!\!+ \; E\}$

$[] \;\hat{=}\; \varnothing$

$[E1, ..., En] \;\hat{=}\; \{(1 \mapsto E1), ..., (n \mapsto En)\}$

Description

Les suites manipulées dans le langage B sont des suites finies. Les suites sont des fonctions totales d'un intervalle entier de la forme $1 .. n$, où $n \in \mathbb{N}$, vers un ensemble quelconque E. Comme les suites sont des fonctions, tous les opérateurs de manipulation des fonctions et donc des relations et des ensembles, sont applicables aux suites. On dit que le n$^{\text{ième}}$ élément d'une suite est la valeur e_n telle que $(n \mapsto e_n)$ appartienne à la suite.

- $\mathsf{seq}\,(E)$ désigne l'ensemble des suites (*sequence* en anglais) dont les éléments appartiennent à l'ensemble E.
- $\mathsf{seq_1}\,(E)$ désigne l'ensemble des suites dans l'ensemble E et qui ne sont pas la suite vide.
- $\mathsf{iseq}\,(E)$ désigne l'ensemble des suites injectives dans l'ensemble E.
- $\mathsf{iseq_1}\,(E)$ désigne l'ensemble des suites injectives dans l'ensemble E et qui ne sont pas la suite vide.
- $\mathsf{perm}\,(E)$ désigne l'ensemble des suites bijectives dans l'ensemble E. Ces suites sont appelées permutations. À noter que l'ensemble E doit être fini.
- $[]$ désigne la suite vide. C'est une fonction qui ne possède pas d'élément. La suite vide n'est autre que l'ensemble vide Ø.
- $[e1, ..., en]$ désigne la suite en extension dont les n éléments sont, dans l'ordre, $e1, ..., en$.

Exemples

Soit l'ensemble $E = \{0, 1, 2\}$,

$[] \in \mathsf{seq}\,(E), [0] \in \mathsf{seq}\,(E), [1, 2, 0] \in \mathsf{seq}\,(E), [0, 2, 2, 0, 1, 0, 0] \in \mathsf{seq}\,(E)$

$[0] \in \mathsf{seq_1}\,(E), [1, 2, 0] \in \mathsf{seq_1}\,(E), [0, 2, 2, 0, 1, 0, 0] \in \mathsf{seq_1}\,(E)$

$[] \in \mathsf{iseq}\,(E), [1] \in \mathsf{iseq}\,(E), [1, 2, 0] \in \mathsf{iseq}\,(E), [0, 2] \in \mathsf{iseq}\,(E)$, mais $[0, 1, 0] \notin \mathsf{iseq}\,(E)$

$[1] \in \mathsf{iseq_1}\,(E), [1, 2, 0] \in \mathsf{iseq_1}\,(E), [0, 2] \in \mathsf{iseq_1}\,(E)$, mais $[0, 1, 0] \notin \mathsf{iseq_1}\,(E)$

$[0, 1, 2] \in \mathsf{perm}\,(E), [1, 0, 2] \in \mathsf{perm}\,(E), [2, 1, 0] \in \mathsf{perm}\,(E)$, mais $[0, 1] \notin \mathsf{perm}\,(E)$

5.18 Expressions de suites

Opérateur

size	Taille
first	Premier élément
last	Dernier élément
front	Tête
tail	Queue
rev	Inverse

Syntaxe

Taille_suite	::=	"size" "(" *Expression* ")"
Premier_élément_suite	::=	"first" "(" *Expression* ")"
Dernier_élément_suite	::=	"last" "(" *Expression* ")"
Tête_suite	::=	"front" "(" *Expression* ")"
Queue_suite	::=	"tail" "(" *Expression* ")"
Inverse_suite	::=	"rev" "(" *Expression* ")"

Règles de typage

Dans les expressions size (S), first (S), last (S), front (S), tail (S) et rev (S), S doit désigner une suite de type $\mathbb{P}(\mathbb{Z}\times T)$. Le type de size (S) est le type entier $\mathbb{Z}$. Le type des expressions first (S) et last (S) est T. Le type des suites front (S), tail (S) et rev (S) est $\mathbb{P}(\mathbb{Z}\times T)$.

Bonne définition

Expression	Condition de bonne définition
size (S)	$S \in \text{seq}(\text{ran}(S))$
first (S)	$S \in \text{seq}_1(\text{ran}(S))$
last (S)	$S \in \text{seq}_1(\text{ran}(S))$
front (S)	$S \in \text{seq}_1(\text{ran}(S))$
tail (S)	$S \in \text{seq}_1(\text{ran}(S))$
rev (S)	$S \in \text{seq}(\text{ran}(S))$

Définitions

$\text{size}([]) \;\hat{=}\; 0$

$\text{size}(S \leftarrow x) \;\hat{=}\; \text{size}(S) + 1$

$\text{first}(S) \;\hat{=}\; S(1)$

$\text{last}(S) \;\hat{=}\; S(\text{size}(S))$

$\text{front}(S) \;\hat{=}\; S \uparrow (\text{size}(S) - 1)$

$\text{tail}(S) \;\hat{=}\; S \downarrow 1$

$\text{rev}(S) \;\hat{=}\; \lambda\, i \,.\, (\, i \in 1..\text{size}(S) \mid S(\,\text{size}(S) - i + 1)\,)$

Description

Soit *S1* une suite, soit *S2* une suite non vide,

- size (*S1*) représente le nombre d'éléments de la suite,
- first (*S2*) représente le premier élément de *S2*,
- last (*S2*) représente le dernier élément de *S2*,
- front (*S2*) représente la suite *S2*, privée de son dernier élément,
- tail (*S2*) représente la suite *S2*, privée de son premier élément,
- rev (*S1*) représente la suite comportant les mêmes éléments que *S1*, mais dans un ordre inverse.

Exemples

Soit la suite *S* = [5, 7, -2, 1],

size (*S*) = 4

first (*S*) = 5

last (*S*) = 1

front (*S*) = [5, 7, -2]

tail (*S*) = [7, -2, 1]

rev (*S*) = [1, -2, 7, 5]

5.19 Expressions de suites (suite)

Opérateur

ˆ	Concaténation
$\rightarrow$	Insertion en tête
$\leftarrow$	Insertion en queue
$\uparrow$	Restriction à la tête
$\downarrow$	Restriction à la queue
conc	Concaténation généralisée

Syntaxe

Concaténation	::=	*Expression* "ˆ" *Expression*
Insertion_tête	::=	*Expression* "→" *Expression*
Insertion_queue	::=	*Expression* "←" *Expression*
Restriction_tête	::=	*Expression* "↑" *Expression*
Restriction_queue	::=	*Expression* "↓" *Expression*
Concat_généralisée	::=	"conc" "(" *Expression* ")"

Règles de typage

Dans l'expression $S1 \mathbin{\hat{}} S2$, $S1$ et $S2$ désignent des suites de type $\mathbb{P}(\mathbb{Z}\times T)$.

Dans les expressions $X \rightarrow S$ et $S \leftarrow X$, S désigne une suite de type $\mathbb{P}(\mathbb{Z}\times T)$ et X désigne un élément de la suite, de type T.

Dans les expressions $S \uparrow n$ et $S \downarrow n$, S désigne une suite de type $\mathbb{P}(\mathbb{Z}\times T)$ et n doit être de type $\mathbb{Z}$.

Dans l'expression conc (S), S désigne une suite dont les éléments sont des suites de même type. S doit donc être de type $\mathbb{P}(\mathbb{Z}\times\mathbb{P}(\mathbb{Z}\times T))$.

Le type des suites $S1 \mathbin{\hat{}} S2$, $X \rightarrow S$, $S \leftarrow X$, $S \uparrow n$, $S \downarrow n$ et conc (S) est $\mathbb{P}(\mathbb{Z}\times T)$.

Bonne définition

Expression	Condition de bonne définition
$S \uparrow n$	n doit appartenir à l'intervalle $0 \,..\, \mathrm{size}\,(S)$
$S \downarrow n$	n doit appartenir à l'intervalle $0 \,..\, \mathrm{size}\,(S)$

Définitions

$S \uparrow n \;\hat{=}\; (1 \,..\, n) \lhd S$

$S \downarrow n \;\hat{=}\; \lambda\, i . (i \in 1 \,..\, \mathrm{size}\,(S) - n \mid S\,(n+i))$

$S1 \mathbin{\hat{}} S2 \;\hat{=}\; S1 \cup \lambda\, i . (\, i \in \mathrm{size}(S1)+1 \,..\, \mathrm{size}(S1)+\mathrm{size}(S2) \mid S2(i - \mathrm{size}(S1))\,)$

$x \rightarrow S \;\hat{=}\; \{1 \mapsto x\} \cup \lambda\, i . (\, i \in 2 \,..\, \mathrm{size}(S)+1 \mid S(i - 1))$

$S \leftarrow x \;\hat{=}\; S \cup \{\mathrm{size}\,(S) + 1 \mapsto x\}$

$\mathrm{conc}\,([]) \;\hat{=}\; []$

$\mathrm{conc}\,(x \rightarrow S) \;\hat{=}\; x \mathbin{\hat{}} \mathrm{conc}\,(S)$

Description

Soient $S1$ et $S2$ des suites,

- $S1 \mathbin{\hat{}} S2$ représente la suite obtenue en concaténant dans l'ordre, les suites $S1$ et $S2$.

Soit S une suite, X un nouvel élément et n un entier relatif,

- $X \rightarrow S$ représente la suite obtenue en insérant en tête de la suite S le nouvel élément X.
- $S \leftarrow X$ représente la suite obtenue en insérant en queue de la suite S le nouvel élément X.
- $S \uparrow n$ représente la suite obtenue à partir de S en ne conservant que ses n premiers éléments.
- $S \downarrow n$ représente la suite obtenue à partir de S en éliminant ses n premiers éléments.

Soit S une suite dont les éléments sont des suites,

- conc (S) représente la suite obtenue en concaténant dans l'ordre toutes les suites qui sont les éléments de S.

Exemples

Soient les suites $S1 = [3, 1]$ et $S2 = [0, -2, 4]$,

$S1 \mathbin{\hat{}} S2 = [3, 1, 0, -2, 4]$

$2 \rightarrow S1 = [2, 3, 1]$

$S1 \leftarrow 2 = [3, 1, 2]$

$S2 \uparrow 2 = [0, -2]$, $S2 \uparrow 4 = [0, -2, 4]$

$S2 \downarrow 2 = [4]$, $S2 \downarrow 3 = []$, $S2 \downarrow 0 = [0, -2, 4]$

Soit la suite $S = [\,[2, 5], [-1, -2, 9], [], [5]\,]$,

conc (S) $= [2, 5, -1, -2, 9, 5]$

5.20 Ensembles d'arbres

Opérateur

tree	Arbres
btree	Arbres binaires

Syntaxe

Arbres	::= "tree" "(" *Expression* ")"
Arbres_binaires	::= "btree" "(" *Expression* ")"

Règles de typage

Dans les expressions tree (S) et btree (S), S doit désigner un ensemble de type $\mathbb{P}(T)$. Le type de tree (S) et de btree (S) est $\mathbb{P}(\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T))$.

Définitions

ins	$\hat{=}\ \lambda i \,.\, (i \in \mathbb{N} \mid \lambda s \,.\, (s \in \text{seq}(\mathbb{N}) \mid i \rightarrow s))$
cns	$\hat{=}\ \lambda t \,.\, (t \in \text{seq}(\mathbb{F}(\text{seq}(\mathbb{N}_1))) \mid \{[]\} \cup \bigcup i \,.\, (i \in \text{dom}(t) \mid \text{ins}(i)[t(i)]))$
tree (S)	$\hat{=}\ \bigcup t \,.\, (t \in \text{T} \mid t \rightarrow S)$
btree (S)	$\hat{=}\ \{ t \mid t \in \text{tree}(S) \wedge \forall n \,.\, (n \in \text{dom}(t) \Rightarrow \text{arity}(t, n) \in \{0, 2\}) \}$

Description

Les arbres modélisés dans le langage B sont des arbres finis non vides décorés et possédant des branches ordonnées.

Un arbre est composé d'un ensemble fini de nœuds. Les nœuds sont reliés par des liens orientés appelés branches. Si une branche relie le nœud A au nœud B, alors on dit que A est le père de B et que B est le fils de A. Un nœud d'un arbre peut avoir au plus un père. Le seul nœud qui n'a pas de père s'appelle la racine de l'arbre. Un arbre n'étant jamais vide, il possède toujours une racine. Un nœud peut avoir de 0 à n fils. On appelle ce nombre l'arité du nœud. Un nœud d'arité nulle s'appelle une feuille. Si un nœud possède un ou plusieurs fils alors leur ordre est significatif, les branches vers les fils sont numérotées de 1 à n.

Un nœud est représenté mathématiquement par la suite des numéros de branches reliant la racine de l'arbre au nœud. La racine de l'arbre est représentée par la suite vide.

L'arbre est dit décoré car un élément d'un ensemble donné S est associé à chacun de ses nœuds.

- tree (S) est l'ensemble des arbres décorés à l'aide d'éléments de l'ensemble S. Un élément de cet ensemble est une fonction totale d'un ensemble fini de nœuds vers un ensemble S. Chaque nœud est représenté par la suite des branches conduisant au nœud à partir de la racine de l'arbre. Une branche d'un nœud est identifiée par un élément de $\mathbb{N}_1$.
- btree (S) est l'ensemble des arbres binaires décorés à l'aide d'éléments de l'ensemble S. Un arbre binaire est un arbre pour lequel l'arité de chaque nœud est égale à 0 ou à 2. Un nœud d'un arbre binaire est donc soit une feuille, soit un nœud à deux branches qu'on appelle branche gauche et branche droite.

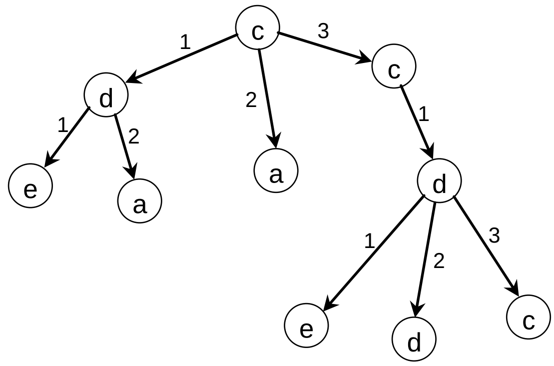
Exemples

Les représentations graphiques des arbres donnés ci-dessous suivent les conventions suivantes. Un nœud est représenté par un cercle qui contient la valeur associée au nœud. Les branches sont représentées par des flèches du père vers le fils, numérotées de 1 à n.

Soit l'ensemble énuméré : $S = \{a, b, c, d, e\}$,

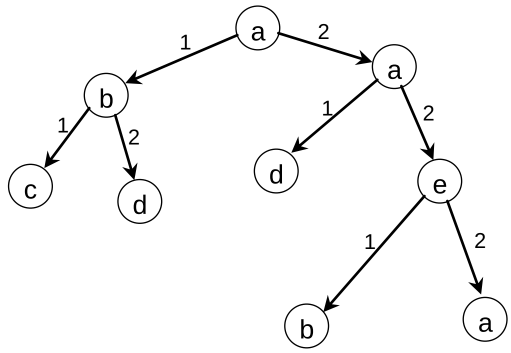
l'arbre A est un élément de tree (S) :

$$\begin{aligned} A = \{\, & [\,] \mapsto c, \\ & [1] \mapsto d, \\ & [1, 1] \mapsto e, \\ & [1, 2] \mapsto a, \\ & [2] \mapsto a, \\ & [3] \mapsto c, \\ & [3, 1] \mapsto d, \\ & [3, 1, 1] \mapsto e, \\ & [3, 1, 2] \mapsto d, \\ & [3, 1, 3] \mapsto c\} \end{aligned}$$



l'arbre B est un élément de btree (S) :

$$\begin{aligned} B = \{\, & [\,] \mapsto a, \\ & [1] \mapsto b, \\ & [1, 1] \mapsto c, \\ & [1, 2] \mapsto d, \\ & [2] \mapsto a, \\ & [2, 1] \mapsto d, \\ & [2, 2] \mapsto e, \\ & [2, 2, 1] \mapsto b, \\ & [2, 2, 2] \mapsto a\} \end{aligned}$$



5.21 Expressions d'arbres

Opérateur

const	Construction
top	Racine
sons	Fils
prefix	Aplatissement préfixé
postfix	Aplatissement postfixé
sizet	Taille
mirror	Symétrie

Syntaxe

Construction_arbre	::=	"const" "(" *Expression* "," *Expression* ")"
Racine_arbre	::=	"top" "(" *Expression* ")"
Fils_arbre	::=	"sons" "(" *Expression* ")"
Aplatissement_préfixé	::=	"prefix" "(" *Expression* ")"
Aplatissement_postfixé	::=	"postfix" "(" *Expression* ")"
Taille_arbre	::=	"sizet" "(" *Expression* ")"
Symétrie_arbre	::=	"mirror" "(" *Expression* ")"

Règles de typage

Dans les expressions $\mathrm{const}\,(x, q)$, $\mathrm{top}\,(t)$, $\mathrm{sons}\,(t)$, $\mathrm{prefix}\,(t)$, $\mathrm{postfix}\,(t)$, $\mathrm{sizet}\,(t)$, $\mathrm{mirror}\,(t)$, t doit être un arbre de type $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$, x doit être de type T et q doit être une suite d'arbres de type $\mathbb{P}(\mathbb{Z} \times \mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T))$. Le type des expressions $\mathrm{const}\,(x, q)$ et $\mathrm{mirror}\,(t)$ est $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$. Le type de $\mathrm{top}\,(t)$ est T. Le type de $\mathrm{sons}\,(t)$ est $\mathbb{P}(\mathbb{Z} \times \mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T))$. Le type des expressions $\mathrm{prefix}\,(t)$ et $\mathrm{postfix}\,(t)$ est $\mathbb{P}(\mathbb{Z} \times T)$. Le type de l'expression $\mathrm{sizet}\,(t)$ est $\mathbb{Z}$.

Bonne définition

Expression	Condition de bonne définition
const (x, q)	$x \in S \;\wedge\; q \in \mathrm{seq}\,(\,\mathrm{tree}\,(S)\,)$
top (t)	$t \in \mathrm{tree}\,(S)$
sons (t)	$t \in \mathrm{tree}\,(S)$
prefix (t)	$t \in \mathrm{tree}\,(S)$
postfix (t)	$t \in \mathrm{tree}\,(S)$
sizet (t)	$t \in \mathrm{tree}\,(S)$
mirror (t)	$t \in \mathrm{tree}\,(S)$

Définitions

$$\mathrm{const}\,(x, q) \;\widehat{=}\; \{\, [\,] \mapsto x \,\} \;\cup\; \bigcup i\,.\,(i \in \mathrm{dom}\,(q) \mid \mathrm{ins}\,(i)^{-1} \;;\; q(i)\,)$$

$$\mathrm{top} \;\widehat{=}\; \mathrm{const}^{-1} \,;\, \mathrm{prj}_1\,(S, \mathrm{seq}\,(\mathrm{tree}\,(S)))$$

$$\mathrm{sons} \;\widehat{=}\; \mathrm{const}^{-1} \,;\, \mathrm{prj}_2\,(S, \mathrm{seq}\,(\mathrm{tree}\,(S)))$$

prefix (t) $\triangleq$ top $(t) \rightarrow$ conc (sons (t) **;** prefix)

postfix (t) $\triangleq$ conc (sons (t) **;** postfix) $\leftarrow$ top (t)

sizet (t) $\triangleq$ succ (sum (sons (t) **;** sizet))

mirror (t) $\triangleq$ const (top (t), rev (sons (t) **;** mirror))

Description

Soit S un ensemble, soit t un arbre décoré par des éléments de S, soit x un élément de S et soit q un élément de seq (seq ($\mathbb{N}_1$) $\nrightarrow$ S)),

- const (x, q) représente l'arbre dont la racine est associée à x et dont les fils sont les éléments de la suite q,
- top (t) représente la valeur associée à la racine de l'arbre t,
- sons (t) représente la suite des fils de la racine de l'arbre t,
- prefix (t) représente l'aplatissement préfixé des éléments de S portés par l'arbre t, dans une suite. Cette suite se définit de manière récursive. Si t est une feuille, prefix (t) est la suite contenant la valeur associée au nœud feuille. Sinon prefix (t) s'obtient en concaténant la suite contenant la valeur associée à la racine de t et les suites préfixées de chacun des fils de la racine de t pris dans l'ordre,
- postfix (t) représente l'aplatissement postfixé des éléments de S portés par l'arbre t, dans une suite. Cette suite se définit de manière récursive. Si t est une feuille, postfix (t) est la suite contenant la valeur associée au nœud feuille. Sinon postfix (t) s'obtient en concaténant les suites postfixées de chacun des fils de la racine de t pris dans l'ordre et la suite contenant la valeur associée à la racine de t,
- sizet (t) représente la taille de l'arbre t. C'est le nombre de nœuds de t. Cette expression se définit de manière récursive,
- mirror (t) représente l'arbre symétrique de l'arbre t. C'est l'arbre construit à partir de t en inversant pour chaque nœud l'ordre des fils du nœud.

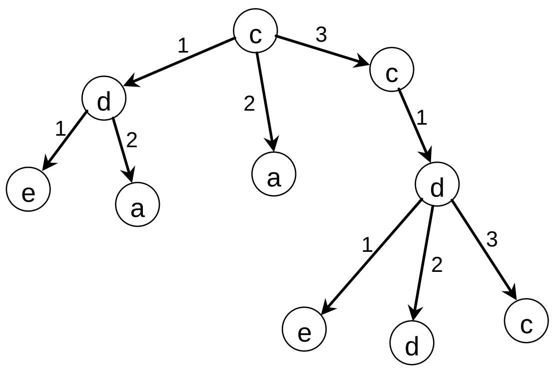
Exemples

Soit l'ensemble énuméré : $S = \{a, b, c, d, e\}$,

Soient $A1$, $A2$ et $A3$ des arbres portant des éléments de S :

$$A1 = \{ [] \mapsto d,\ [1] \mapsto e,\ [2] \mapsto a \}$$
$$A2 = \{ [] \mapsto a \}$$
$$A3 = \{ [] \mapsto c,\ [1] \mapsto d,\ [1, 1] \mapsto e,\ [1, 2] \mapsto d,\ [1, 3] \mapsto c \}$$

$A = \text{const}\ (c, [A1, A2, A3])$



Alors,

$$A = \{ [] \mapsto c,$$
$$[1] \mapsto d,\ [1, 1] \mapsto e,\ [1, 2] \mapsto a,$$
$$[2] \mapsto a,$$
$$[3] \mapsto c,\ [3, 1] \mapsto d,\ [3, 1, 1] \mapsto e,\ [3, 1, 2] \mapsto d,\ [3, 1, 3] \mapsto c \}$$

top $(A) = c$

sons $(A) = [A1, A2, A3]$

prefix $(A) = [c, d, e, a, a, c, d, e, d, c]$

postfix $(A) = [e, a, d, a, e, d, c, d, c, c]$

$\text{sizet}(A) = 10$

$\text{mirror}(A) =$

$\{ \quad [] \mapsto c,$

$[1] \mapsto c,\ [1, 1] \mapsto d,\ [1, 1, 1] \mapsto c,\ [1, 1, 2] \mapsto d,\ [1, 1, 3] \mapsto e,$

$[2] \mapsto a,$

$[3] \mapsto d,\ [3, 1] \mapsto a,\ [3, 2] \mapsto e \}$

5.22 Expressions de nœuds d'arbres

Opérateur

rank	Rang d'un nœud
father	Père d'un nœud
son	Fils d'un nœud
subtree	Sous-arbre
arity	Arité

Syntaxe

Rang_noeud	::=	"rank" "(" *Expression* "," *Expression* ")"
Père_noeud	::=	"father" "(" *Expression* "," *Expression* ")"
Fils_noeud	::=	"son" "(" *Expression* "," *Expression* "," *Expression* ")"
Sous_arbre_noeud	::=	"subtree" (" *Expression* "," *Expression* ")"
Arité_noeud	::=	"arity" "(" *Expression* "," *Expression* ")"

Règles de typage

Dans les expressions rank (t, n), father (t, n), son (t, n, i), subtree (t, n), arity (t, n), t doit être un arbre de type $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$, n doit être une suite de type $\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$ et i doit être de type $\mathbb{Z}$. Le type des expressions rank (t, n) et arity (t, n) est le type entier $\mathbb{Z}$. Le type des expressions father (t, n), son (t, n, i) est $\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$. Le type de subtree (t, n) est $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$.

Bonne définition

Expression	Condition de bonne définition
rank (t, n)	$t \in$ tree (S) $\wedge$ $n \in$ dom $(t) - \{[]\}$
father (t, n)	$t \in$ tree (S) $\wedge$ $n \in$ dom $(t) - \{[]\}$
son (t, n, i)	$t \in$ tree (S) $\wedge$ $n \leftarrow i \in$ dom (t)
subtree (t, n)	$t \in$ tree (S) $\wedge$ $n \in$ dom (t)
arity (t, n)	$t \in$ tree (S) $\wedge$ $n \in$ dom (t)

Définitions

rank (t, n)	$\triangleq$	last (n)
father (t, n)	$\triangleq$	front (n)
son (t, n, i)	$\triangleq$	$n \leftarrow i$
subtree (t, n)	$\triangleq$	$\lambda u . (u \in$ seq $(S) \mid n \mathbin{\hat{}} u)$; t
arity (t, n)	$\triangleq$	size (sons (subtree (t, n)))

Description

Soit S un ensemble, soit t un arbre décoré par des éléments de S, soient n et m des nœuds de t, (sous la forme d'une suite de numéros de branches décrivant le chemin vers le nœud à partir de la racine). Le nœud m ne doit pas être la racine de l'arbre. Enfin, soit i l'une des branches partant de n.

- rank (t, m) représente le rang de la branche reliant le père de m à m,

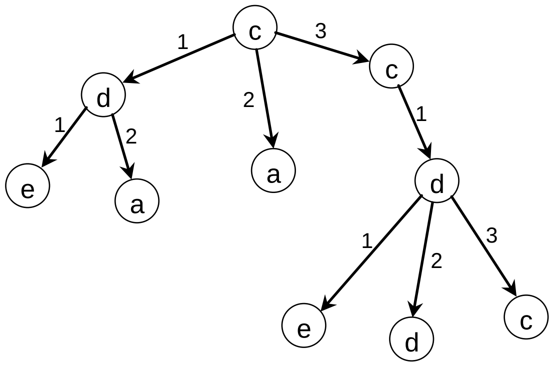
- father (t, n) représente le père du nœud n dans l'arbre t,
- son (t, n, i) représente le fils de rang i du nœud n de l'arbre t,
- subtree (t, n) représente le sous-arbre de l'arbre t dont la racine est le nœud n,
- arity (t, n) représente l'arité du nœud n dans l'arbre t, c'est-à-dire le nombre de fils de n.

Exemples

Soit l'ensemble énuméré : $S = \{a, b, c, d, e\}$,

Soit l'arbre A portant des éléments de S :

$$
\begin{aligned}
A = \{ & [] \mapsto c, \\
& [1] \mapsto d,\ [1, 1] \mapsto e,\ [1, 2] \mapsto a, \\
& [2] \mapsto a, \\
& [3] \mapsto c,\ [3, 1] \mapsto d,\ [3, 1, 1] \mapsto e, \\
& [3, 1, 2] \mapsto d,\ [3, 1, 3] \mapsto c \}
\end{aligned}
$$



Alors,

rank $(A, [3, 1, 2]) = 2$

father $(A\ [3, 1, 2]) = [3, 1]$

son $(A, [3, 1], 2) = [3, 1, 2]$

subtree $(A, [3, 1]) = \{ [] \mapsto d, [1] \mapsto e, [2] \mapsto d, [3] \mapsto c \}$

arity $(A, [1]) = 2$

5.23 Expressions d'arbres binaires

Opérateur

bin	Arbre binaire en extension
left	Sous-arbre gauche
right	Sous-arbre droit
infix	Aplatissement infixé

Syntaxe

Arbre_binaire_en_extension	::=	"bin" "(" *Expression* ["," *Expression* "," *Expression*] ")"
Sous_arbre_gauche	::=	"left" "(" *Expression* ")"
Sous_arbre_droit	::=	"right" "(" *Expression* ")"
Aplatissement_infixé	::=	"infix" "(" *Expression* ")"

Règles de typage

Dans les expressions bin (x), bin (g, x, d), left (t), right (t), infix (t), x est de type T, g et d doivent être des arbres de type $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$, t doit être un arbre de type $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$. Le type des expressions bin (x), bin (g, x, d), left (t), right (t) est le type $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$. Le type de infix (t) est $\mathbb{P}(\mathbb{Z} \times T)$.

Bonne définition

Expression	Condition de bonne définition
bin (x)	$x \in S$
bin (g, x, d)	$x \in S \;\wedge\; g \in$ btree $(S) \;\wedge\; d \in$ btree (S)
left (t)	$t \in$ btree $(S) \;\wedge\;$ sons $(t) \neq []$
right (t)	$t \in$ btree $(S) \;\wedge\;$ sons $(t) \neq []$
infix (t)	$t \in$ btree (S)

Définitions

bin (x)	$\triangleq$	const $(x, [])$
bin (g, x, d)	$\triangleq$	const $(x, [g, d])$
left (t)	$\triangleq$	first (sons (t))
right (t)	$\triangleq$	last (sons (t))
infix (bin (x))	$\triangleq$	$[x]$
infix (bin (g, x, d))	$\triangleq$	infix (g) ˆ $[x]$ ˆ infix (d)

Description

Les arbres binaires étant des arbres (cf. §5.20 *Ensembles d'arbres*), toutes les expressions d'arbres et les expressions de nœuds d'arbres des sections précédentes peuvent s'appliquer aux arbres binaires. Les expressions décrites ci-dessous sont spécifiques aux arbres binaires.

Soit S un ensemble, soit x un élément de S, g, d, t et u des arbres binaires décorés par des éléments de S, où t n'est pas une feuille.

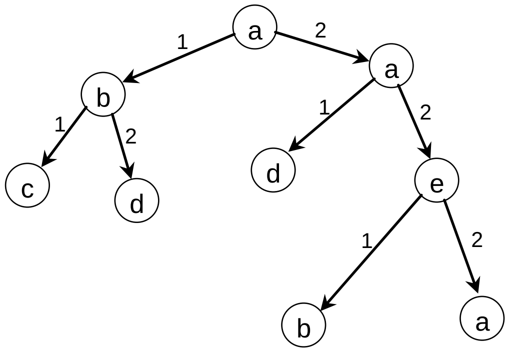
- bin (x) représente l'arbre binaire composé d'un seul nœud portant la valeur x,
- bin (g, x, d) représente l'arbre binaire dont la racine porte la valeur x, et dont les fils gauche et droit de la racine sont les arbres g et d,
- left (t) représente le sous-arbre gauche de l'arbre t,
- right (t) représente le sous-arbre droit de l'arbre t,
- infix (u) représente l'aplatissement infixé des éléments de S portés par l'arbre u, dans une suite. Cette suite se définit de manière récursive. Si u est une feuille, infix (u) est la suite contenant la valeur associée au nœud feuille. Sinon infix (u) s'obtient en concaténant la suite infixée du sous-arbre gauche de u, la suite contenant la valeur associée à la racine de u et la suite infixée du sous-arbre droit de u.

Exemples

Soit l'ensemble énuméré : $S = \{a, b, c, d, e\}$,

Soit l'arbre B portant des éléments de S :

$$\begin{aligned} B = \{ & [] \mapsto a, \\ & [1] \mapsto b, \\ & [1, 1] \mapsto c, \\ & [1, 2] \mapsto d, \\ & [2] \mapsto a, \\ & [2, 1] \mapsto d, \\ & [2, 2] \mapsto e, \\ & [2, 2, 1] \mapsto b, \\ & [2, 2, 2] \mapsto a\} \end{aligned}$$



B = bin (bin (bin (c), b, bin (d)),
a,
bin (bin (d), a, bin (bin (b), e, bin (a))))

left $(B) = \{ [] \mapsto b,\ \ [1] \mapsto c,\ \ [2] \mapsto d \}$

right $(B) = \{ [] \mapsto a,\ \ [1] \mapsto d,\ \ [2] \mapsto e,\ \ [2, 1] \mapsto b,\ \ [2, 2] \mapsto a \}$

infix $(B) = [c, b, d, a, d, a, b, e, a]$

6 SUBSTITUTIONS

Syntaxe

Substitution ::=
Substitution_bloc
| *Substitution_identité*
| *Substitution_devient_égal*
| *Substitution_précondition*
| *Substitution_assertion*
| *Substitution_choix_borné*
| *Substitution_conditionnelle*
| *Substitution_sélection*
| *Substitution_cas*
| *Substitution_choix_non_borné*
| *Substitution_définition_locale*
| *Substitution_devient_elt_de*
| *Substitution_devient_tel_que*
| *Substitution_variable_locale*
| *Substitution_séquence*
| *Substitution_appel_opération*
| *Substitution_simultanée*
| *Substitution_tant_que*

Description

Les substitutions sont des notations mathématiques permettant de modéliser la transformation de prédicats, définis dans les chapitres précédents.

Soient S une substitution et P un prédicat. Alors, la notation :

$[\, S\,]\, P$ (lire « la substitution S établit le prédicat P »)

représente le prédicat obtenu après transformation de P par la substitution S. Le vocabulaire suivant est également employé pour désigner cette transformation : on parle de l'établissement par la substitution S de la postcondition P. On parle aussi de l'application de la substitution S à P. Les substitutions modélisent l'aspect dynamique des modules B : l'initialisation et les opérations, puisqu'elles permettent d'établir comment les propriétés portant sur les données du module sont transformées par ses opérations.

Exemple

Voici une forme simple de la substitution « devient égal ». Soient x et y des variables entières, alors :

$[\, x := 3\,]\, (\, y + x < 0\,)$

désigne le prédicat obtenu après le remplacement dans le prédicat $y + x < 0$ de toutes les occurrences libres de la variable x par l'expression 3. On obtient alors le prédicat :

$y + 3 < 0$

Ainsi l'application de cette substitution « devient égal » correspond bien à l'application d'une substitution en ce sens que la valeur de x dans le prédicat $y + x < 0$ est remplacée par 3.

Substitutions généralisées

L'ensemble des substitutions utilisables dans le langage B est décrit par le langage des substitutions généralisées. Chaque substitution généralisée se définit en précisant quel est le prédicat obtenu après application de la substitution à un prédicat quelconque.

Dans les chapitres suivants, on donne la description détaillée des substitutions généralisées. En voici la liste :

Opérateur ou mot réservé	Nom de la production grammaticale
BEGIN	Substitution bloc
skip	Substitution identité
:=	Substitution devient égal
:( )	Substitution devient tel que
:∈	Substitution devient élément de
PRE	Substitution précondition
ASSERT	Substitution assertion
CHOICE	Substitution choix borné
IF	Substitution conditionnelle
SELECT	Substitution sélection
CASE	Substitution cas
ANY	Substitution choix non borné
LET	Substitution définition locale
VAR	Substitution variable locale
;	Substitution séquence
WHILE	Substitution tant que
←	Substitution appel d'opération
||	Substitution simultanée

Utilisation des substitutions

Les substitutions généralisées sont utilisées dans le Langage B afin de décrire le corps de l'initialisation et des opérations d'un composant (cf. §7.22 *La clause INITIALISATION* et §7.23 *La clause OPERATIONS*). Le mécanisme de transformation d'un prédicat par une substitution permet de générer de manière systématique les obligations de preuves concernant l'initialisation et les opérations. Par exemple, pour qu'une machine abstraite soit sémantiquement correcte, il faut démontrer que chaque opération de la machine préserve l'invariant. Pour ce faire, on génère une obligation de preuve avec comme hypothèse (parmi d'autres) l'invariant de la machine et comme but le prédicat obtenu après transformation de l'invariant par la substitution définissant l'opération. De même on engendre des obligations de preuves qui permettent de démontrer l'établissement par l'initialisation de l'invariant et la conservation de la spécification d'une opération lors de son raffinement.

Non déterminisme

Une substitution possède un comportement non déterministe si elle décrit plusieurs comportements possibles sans préciser lequel sera effectivement choisi.

En B, les substitutions des machines et des raffinements peuvent être non déterministes. Le non déterminisme décroît lors du raffinement. Les substitutions des implantations doivent être déterministes.

Dans les sections suivantes, les substitutions généralisées sont présentées les unes après les autres. Pour chaque substitution, on donne son nom, sa syntaxe, éventuellement ses règles de typage, ses règles de portée, ses restrictions sémantiques, ses conditions de bonne définition, sa description et un exemple.

6.1 Substitution bloc

Syntaxe

Substitution_bloc ::= "BEGIN" *Substitution* "END"

Définition

Soient S une substitution et P un prédicat, alors :

$$\text{BEGIN } S \text{ END} \;\hat{=}\; S$$

Description

La substitution bloc parenthèse une substitution. On peut ainsi grouper dans un seul bloc plusieurs substitutions réalisées en séquence ou en parallèle.

Exemples

```
BEGIN
    x := x + 1 ;
    y := x²
END
```

6.2 Substitution identité

Syntaxe

Substitution_identité ::= "skip"

Définition

Soit P un prédicat, alors :

[skip] $P \quad \Leftrightarrow \quad P$

Description

La substitution identité ne modifie pas le prédicat sur lequel elle est appliquée. Elle est notamment utilisée pour décrire que certaines branches d'une substitution IF, CASE ou SELECT ne modifient pas les variables.

6.3 Substitution devient égal

Syntaxe

Substitution_devient_égal ::=
    Ident_ren$^{+","}$ ":=" *Expression*$^{+","}$
| *Ident_ren* "(" *Expression*$^{+","}$ ")" ":=" *Expression*
| *Ident_ren* ("'" Ident)$^{+}$ ":=" *Expression*

Définitions

1. Soit x une variable, e une expression et P un prédicat, alors :

   $$[\, x := e \,]\, P$$

   est le prédicat obtenu en remplaçant toutes les occurrences libres de x dans P par e. La notion d'occurrence libre ou liée est définie dans [B-Book] §1.3.3.

2. Soient x et y des variables modifiables, E et F des expressions de même type que x et y et P un prédicat. Enfin, soit z une variable intermédiaire différente de x et de y et *non libre* dans E, F et P. Alors :

   $$[x, y := E, F]\, P \quad \Leftrightarrow \quad [z := F][x := E][y := z]\, P$$

   La construction d'une substitution « devient égal » multiple pour une liste de plus de deux variables se définit alors de manière itérative.

3. Soit f une fonction, x et y des expressions et P un prédicat. Alors :

   $$[f(x) := y]\, P \quad \Leftrightarrow \quad [f := f \mathbin{<\!\!+} \{x \mapsto y\}]\, P$$

4. Soit n un entier supérieur ou égal à 1 et i un entier compris entre 1 et n. Soient rc une donnée record de type $\mathrm{struct}\,(Ident_1 : T_1, \ldots, Ident_n : T_n)$, y une expression et P un prédicat. Alors :

   $$[rc'Ident_i := y]P \quad \Leftrightarrow \quad [rc := \mathrm{rec}\,(Ident_1 : rc'Ident_1, \ldots, Ident_i : y, \ldots, Ident_n : rc'Ident_n)]P$$

   Cette définition s'étend aux accès imbriqués à des champs de records. Nous donnons la définition pour deux niveaux d'accès. Soient rc une donnée record de type $\mathrm{struct}\,(c_1^1 : T_1^1, \ldots, c_i^1 : \mathrm{struct}\,(c_1^2 : T_1^2, \ldots, c_j^2 : T_j^2, \ldots, c_m^2 : T_m^2), \ldots, c_n^1 : T_n^1)$, y une expression et P un prédicat. Alors :

   $$[rc'c_i^1'c_j^2 := y]P \quad \Leftrightarrow \quad [rc := \mathrm{rec}\,(c_1^1 : rc'c_1^1, \ldots, c_i^1 : \mathrm{rec}\,(c_1^2 : rc'c_i^1'c_1^2, \ldots, c_j^2 : y, \ldots, c_m^2 : rc'c_i^1'c_m^2), \ldots, c_n^1 : rc'c_n^1)]P$$

Règles de typage

Dans la substitution $x := e$, x et e doivent être du même type T.

Dans la substitution $x1, \ldots, xn := e1, \ldots, en$, chaque xi doit être du même type que ei.

Dans la substitution $f(x_1, \ldots, x_n) := e$, f doit être de type $\mathbb{P}(T_1 \times \ldots \times T_n \times T_0)$. Alors, chaque x_i doit être de type T_i et e doit être de type T_0.

Dans la substitution $rc'Ident_i := y$, rc doit être de type $\mathrm{struct}\,(Ident_1 : T_1, \ldots, Ident_n : T_n)$. Alors, y doit être de type T_i. Cette règle s'étend aux accès imbriqués de champs de records. Dans le cas de deux accès, la règle devient : dans la substitution $rc'c_i^1'c_j^2 := y$, rc doit être de type $\mathrm{struct}\,(c_1^1 : T_1^1, \ldots, c_i^1 : \mathrm{struct}\,(c_1^2 : T_1^2, \ldots, c_j^2 : T_j^2, \ldots, c_m^2 : T_m^2), \ldots, c_n^1 : T_n^1)$. Alors, y doit être de type T_j^2.

Restrictions

1. Dans le cas d'une substitution « devient égal » d'une liste de variables, les variables doivent être deux à deux distinctes.
2. Dans le cas d'une substitution « devient égal » d'une liste de variables par une liste d'expressions, le nombre des variables doit être identique au nombre des expressions.
3. Chaque variable débutant une substitution « devient égal » doit être une donnée modifiable, c'est-à-dire accessible en écriture selon les règles de visibilité données en Annexe C *Tables de visibilité*.

Description

La substitution « devient égal » remplace une variable par une expression. Elle est définie sous plusieurs formes :

1. substitution « devient égal » pour une variable

La substitution $x := E$ remplace les occurrences de la variable x par l'expression E.

2. substitution « devient égal » pour une liste de variables

La substitution « devient égal » multiple, pour une liste de variables. Une telle substitution multiple correspond à une liste de substitutions devient égal pour une variable simples effectuées simultanément.

3. substitution « devient égal » pour un élément de fonction

La substitution « devient égal » pour un élément de fonction est une abréviation pour remplacer un élément d'une fonction par une expression. La notation, $f(x) := y$, désigne en fait la substitution « devient égal » de f avec elle-même, surchargée pour l'élément d'indice x par la valeur de l'expression y.

4. substitution « devient égal » pour un record

La substitution « devient égal » pour un champ d'un record est une abréviation pour le remplacement d'un champ d'une variable record par une expression. La notation $rc'Ident_i$ désigne en fait la substitution « devient égal » de rc avec une expression donnée record en extension de même type que rc dont la valeur du champ $Ident_i$ vaut y et dont la valeur des autres champs reste inchangée. Dans le cas où une variable record contient un champ record, on étend cette définition récursivement, à l'accès imbriqué de plusieurs champs de la variable record.

Exemples

$x := y + 1$;
$tab := \{(0 \mapsto 3), (1 \mapsto 1), (2 \mapsto -7)\}$;
$tab(1) := 12$;
$tab2 := tab3$;
$u, v, w := 0, 0, 0$;
$p, q := q, p$;
$tab4(x + 2) := 1$;
$rc'c2 :=$ FALSE ;
$rdv'Date'Jour := 13$

6.4 Substitution précondition

Syntaxe

Substitution_précondition ::= "PRE" *Prédicat* "THEN" *Substitution* "END"

Définition

Soient P et R des prédicats et S une substitution.

$$[\text{PRE } P \text{ THEN } S \text{ END}]\, R \Leftrightarrow P \wedge [S]\, R$$

Restriction

1. La substitution précondition n'est pas une substitution d'implantation.

Description

La substitution précondition fixe les préconditions sous lesquelles une opération est appelée.

L'obligation de preuve de conservation de l'invariant I d'une opération définie par une substitution précondition PRE P THEN S END est la suivante : $I \Rightarrow [\text{PRE } P \text{ THEN } S \text{ END}]\, I$

On peut cependant ajouter l'hypothèse P à I puisque, par définition, une telle opération ne peut être appelée que sous sa précondition (cf. paragraphe suivant). L'obligation de preuve devient dès lors : $(I \wedge P) \Rightarrow [\text{PRE } P \text{ THEN } S \text{ END}]\, I$

c'est-à-dire en définitive : $(I \wedge P) \Rightarrow [S]\, I$

Lors de l'appel d'une opération possédant une précondition PRE P THEN S END, l'application de la substitution précondition correspond à la preuve de la précondition P et à l'application de la substitution S. Si la précondition n'est pas prouvée, alors la substitution ne se termine pas. Autrement dit, le comportement décrit par une substitution avec précondition n'est garanti que si, dans son contexte d'utilisation, la précondition est vraie.

Il faut bien distinguer la substitution précondition de la substitution conditionnelle IF. La première n'est utilisable que si le prédicat est valide, alors que la seconde est toujours réalisée, mais son résultat dépend de la validité d'un prédicat.

Exemple

```
PRE
    x ∈ NAT₁
THEN
    x := x - 1
END
```

6.5 Substitution assertion

Syntaxe

Substitution_assertion ::= "ASSERT" *Prédicat* "THEN" *Substitution* "END"

Définition

Soient P et R des prédicats et S une substitution.

$$[\text{ASSERT } P \text{ THEN } S \text{ END}]\, R \;\Leftrightarrow\; P \wedge (P \Rightarrow [S]\, R)$$

Description

La substitution assertion ASSERT P THEN S END permet d'appliquer la substitution S sous l'assertion que le prédicat P est vrai. Cette substitution est très proche de la substitution précondition. Comme dans le cas de la précondition, si le prédicat P n'est pas établi, alors la substitution échoue. Cependant elle présente l'intérêt de mettre en hypothèse P pour l'application de la substitution S, ainsi que pour les substitutions qui suivent S jusqu'à la fin du corps de l'opération ou de l'initialisation dans laquelle elle est employée.

Le rôle des substitutions précondition et assertion diffère. L'utilisation principale d'une précondition est de typer et d'exprimer des propriétés concernant les paramètres d'entrée d'une opération alors que l'utilisation d'une substitution assertion est de fournir au sein d'une opération des hypothèses qui pourront faciliter la preuve de l'opération.

La substitution assertion peut s'avérer utile lors de la preuve du raffinement d'une opération contenant des structures conditionnelles. Si l'opération et l'opération raffinée contiennent toutes les deux des substitutions IF ayant des conditions équivalentes, alors indiquer cette équivalence dans une substitution assertion permet de montrer immédiatement que les obligations de preuve portant sur des cas de raffinement croisés (cas où on a en hypothèse la condition d'un des IF et la négation de la condition de l'autre IF) sont trivialement fausses. En contrepartie, il faut établir l'assertion.

Exemple

```
ASSERT
    x < 5  ⇔  y = 0
THEN
    x := x - 5
END
```

6.6 Substitution choix borné

Syntaxe

Substitution_choix_borné ::= "CHOICE" *Substitution* ("OR" *Substitution*)* "END"

Définition

Soient $S1$, ..., Sn des substitutions (avec $n \geq 2$) et P un prédicat. Alors la substitution CHOICE se définit par :

$$[\text{CHOICE } S1 \text{ OR } ... \text{ OR } Sn \text{ END}]\ P \;\Leftrightarrow\; [S1]\ P \wedge ... \wedge [Sn]\ P$$

Restriction

1. La substitution choix borné n'est pas une substitution d'implantation.

Description

La substitution choix borné permet de définir un nombre fini de comportements possibles sans préciser lequel sera effectivement implanté. Elle définit donc un comportement non déterministe.

Exemple

```
CHOICE
    x1 := x1 + 1
OR
    x1 := x1 - 1
END
```

6.7 Substitution conditionnelle IF

Syntaxe

Substitution_conditionnelle ::=
"IF" *Prédicat* "THEN" *Substitution*
("ELSIF" *Prédicat* "THEN" *Substitution*)*
["ELSE" *Substitution*]
"END"

Définition

Soient $P1$, $P2$, ..., Pn et R des prédicats (avec $n \geq 1$) et soient $S1$, $S2$, ..., Sn et T des substitutions, alors :

1. $[\text{IF } P1 \text{ THEN } S1 \text{ ELSE } T \text{ END}]R \;\Leftrightarrow\; (P1 \Rightarrow [S1]R) \wedge (\neg P1 \Rightarrow [T]R)$
2. $\text{IF } P1 \text{ THEN } S1 \text{ END} \;\hat{=}\; \text{IF } P1 \text{ THEN } S1 \text{ ELSE skip END}$
3. $[\text{IF } P1 \text{ THEN } S1 \text{ ELSIF } P2 \text{ THEN } S2 \text{ ... ELSIF } Pn \text{ THEN } Sn \text{ ELSE } T \text{ END}]R \;\Leftrightarrow$
   $(P1 \Rightarrow [S1]R) \wedge ((\neg P1 \wedge P2) \Rightarrow [S2]R) \wedge ... \wedge ((\neg P1 \wedge ... \wedge \neg Pn\text{-}1 \wedge Pn) \Rightarrow [Sn]R) \wedge ((\neg P1 \wedge ... \wedge \neg Pn) \Rightarrow [T]R)$
4. $\text{IF } P1 \text{ THEN } S1 \text{ ELSIF } P2 \text{ THEN } S2 \text{ ... ELSIF } Pn \text{ THEN } Sn \text{ END} \;\hat{=}$
   $\text{IF } P1 \text{ THEN } S1 \text{ ELSIF } P2 \text{ THEN } S2 \text{ ... ELSIF } Pn \text{ THEN } Sn \text{ ELSE skip END}$

Description

La substitution conditionnelle IF définit plusieurs comportements en fonction de la validité d'un ou de plusieurs prédicats. Le comportement défini par la substitution conditionnelle IF est déterministe. La substitution conditionnelle IF est définie selon plusieurs formes :

1. IF *P1* THEN *S1* ELSE *T* END
   Si le prédicat *P1* est vrai alors la substitution *S1* s'applique, sinon la substitution *T* s'applique.
2. IF *P1* THEN *S1* END
   La branche ELSE d'une substitution IF est facultative. Si elle est absente, elle représente par défaut la substitution identité.
3. IF *P1* THEN *S1* ELSIF *P2* THEN *S2* ... ELSIF *Pn* THEN *Sn* ELSE *T* END
   La présence d'une branche ELSIF dans une substitution IF équivaut à imbriquer une autre substitution IF dans la branche ELSE du premier IF. Il est possible d'avoir un nombre quelconque de branches ELSIF dans une même substitution IF.
4. IF *P1* THEN *S1* ELSIF *P2* THEN *S2* ... ELSIF *Pn* THEN *Sn* END
   Lorsqu'une substitution IF possède un nombre quelconque de branches ELSIF et pas de branche ELSE explicite, cette substitution est définie par défaut avec une branche ELSE contenant la substitution identité.

Exemples

```
IF x ∈ { 2, 4, 8 } THEN
    x := x / 2
END ;
```

```
IF  y + z < 0  THEN
        y := - z
ELSE
        y := 0
END  ;

IF  v = 0  THEN
        signe := 0
ELSIF  v > 0  THEN
        signe := 1
ELSE
        signe := -1
END
```

6.8 Substitution sélection

Syntaxe

Substitution_sélection ::=
"SELECT" *Prédicat* "THEN" *Substitution*
("WHEN" *Prédicat* "THEN" *Substitution*)*
["ELSE" *Substitution*]
"END"

Définition

Soient $P1$, $P2$, ..., Pn et R des prédicats, avec $n \geq 1$. Soient $S1$, $S2$, ..., Sn et T des substitutions, alors :

1. [SELECT $P1$ THEN $S1$ WHEN $P2$ THEN $S2$ ... WHEN Pn THEN Sn END] R $\Leftrightarrow$
   $(P1 \Rightarrow [S1]\ R) \wedge (P2 \Rightarrow [S2]\ R) \wedge ... \wedge (Pn \Rightarrow [\ Sn\ ]\ R)$
2. [SELECT $P1$ THEN $S1$ WHEN $P2$ THEN $S2$ ... WHEN Pn THEN Sn ELSE T END] R $\Leftrightarrow$
   $(P1 \Rightarrow [S1]\ R) \wedge (P2 \Rightarrow [S2]\ R) \wedge ... \wedge (Pn \Rightarrow [Sn]\ R) \wedge ((\neg P1 \wedge \neg P2 \wedge ... \wedge \neg Pn) \Rightarrow [T]\ R)$
3. [SELECT $P1$ THEN $S1$ END] R $\Leftrightarrow$ $P1 \Rightarrow [S1]\ R$
4. [SELECT $P1$ THEN $S1$ ELSE T END] R $\Leftrightarrow$ $P1 \Rightarrow [S1]\ R \ \wedge\ (\neg P1 \Rightarrow [T]\ R)$

Restriction

1. La substitution SELECT n'est pas une substitution d'implantation.

Description

La substitution SELECT définit pour un programme différents comportements possibles en fonction de la validité de prédicats. Chaque branche de la substitution SELECT décrit l’un de ces cas. La branche comporte un prédicat et une substitution. Si le prédicat de cette branche est vrai, alors la substitution peut s’appliquer. Si tous les prédicats sont faux et que la substitution SELECT se termine par une branche ELSE, alors la substitution de la branche ELSE s’applique.

Si les prédicats des différentes branches ne s’excluent pas mutuellement, plusieurs comportements sont possibles et il n’est pas précisé lequel sera effectivement implanté. Dans ce cas le comportement de la substitution SELECT est non déterministe. D’autre part, si aucun des prédicats n’est valide et si la branche ELSE n’existe pas, alors la substitution est non implémentable.

Exemple

SELECT $x \geq 0$ THEN
    $y := x^2$
WHEN $x \leq 0$ THEN
    $y := - x^2$
END

6.9 Substitution condition par cas

Syntaxe

Substitution_cas ::=
"CASE" *Expression* "OF"
"EITHER" *Terme_simple*$^{+","}$ "THEN" *Substitution*
("OR" *Terme_simple*$^{+","}$ "THEN" *Substitution*)*
["ELSE" *Substitution*]
"END"
"END"

Définitions

Soient E une expression, $L1$, $L2$, ..., Ln des listes de constantes littérales distinctes, avec $n \geq 1$. Soient $S1$, $S2$, ..., Sn et T des substitutions, alors :

1. CASE E OF EITHER $L1$ THEN $S1$ OR $L2$ THEN $S2$ ... OR Ln THEN Sn END END $\hat{=}$
   SELECT $E \in \{L1\}$ THEN $S1$ WHEN $E \in \{L2\}$ THEN $S2$ WHEN ... WHEN $E \in \{Ln\}$ THEN Sn
   ELSE skip END
2. CASE E OF EITHER $L1$ THEN $S1$ OR $L2$ THEN $S2$ ... OR Ln THEN Sn ELSE T END END $\hat{=}$
   SELECT $E \in \{L1\}$ THEN $S1$ WHEN $E \in \{L2\}$ THEN $S2$ WHEN ... WHEN $E \in \{Ln\}$ THEN Sn
   ELSE T END

Règle de typage

Dans une substitution CASE, l'expression ainsi que les listes de constantes des branches EITHER et OR doivent toutes être du même type. Ce type doit être un type de base sauf le type STRING.

Restriction

1. Les termes simples de chaque branche EITHER et OR doivent être des constantes littérales (entiers littéraux, énumérés littéraux ou booléens littéraux) deux à deux distinctes, telles que chaque branche soit incompatible avec les autres.

Description

La substitution CASE permet de définir pour un programme différents comportements possibles en fonction de la valeur d'une expression. Chaque branche EITHER et OR est constituée d'une liste non vide de constantes littérales. Si la valeur de l'expression appartient à l'une des branches, alors la substitution de cette branche est exécutée. Sinon la substitution de la branche ELSE est appliquée, si cette dernière branche est absente, elle réalise par défaut la substitution identité. Le comportement de cette substitution est donc déterministe et toujours faisable.

Exemple

```
CASE x / 10 OF
    EITHER 0 THEN
        x := 0
    OR 2, 4, 8 THEN
        x := 1
    OR 3, 9 THEN
        x := 2
    ELSE
        x := -1
    END
END
```

6.10 Substitution choix non borné

Syntaxe

Substitution_choix_non_borné ::= "ANY" Ident$^{+","}$ "WHERE" *Prédicat* "THEN" *Substitution* "END"

Définition

Soient *X* une liste non vide de variables deux à deux distinctes, *S* une substitution et *P* et *R* deux prédicats, alors :

$$[\text{ANY } X \text{ WHERE } P \text{ THEN } S \text{ END}]\, R \;\Leftrightarrow\; \forall X . (P \Rightarrow [S]\, R)$$

Règle de portée

Dans une substitution ANY *X* WHERE *P* THEN *S* END, la portée de la liste d'identificateurs *X* est le prédicat *P* et la substitution *S*.

Restrictions

1. La substitution ANY n'est pas une substitution d'implantation.
2. Les identificateurs introduits dans une substitution ANY doivent être deux à deux distincts.
3. Les variables *X* introduites par la substitution ANY *X* WHERE *P* THEN *S* END doivent être typées par un prédicat de typage de données abstraites (cf. §3.3 *Typage des données abstraites*), situé dans une liste de conjonctions au plus haut niveau d'analyse syntaxique de *P*. Ces variables ne peuvent pas être utilisées dans *P* avant d'avoir été typées.

Description

La substitution ANY *X* WHERE *P* THEN *S* END permet d'utiliser dans la substitution *S* les données abstraites déclarées dans la liste *X* et vérifiant le prédicat *P*.

Si plusieurs valeurs satisfont le prédicat *P* la substitution définit alors un comportement non déterministe. Les données abstraites de la liste *X* sont accessibles en lecture, mais pas en écriture dans *S*, car ce ne sont pas des variables locales mais des données abstraites définies par le prédicat *P*.

Exemple

```
ANY r1, r2 WHERE
    r1 ∈ NAT ∧
    r2 ∈ NAT ∧
    r1² + r2² = 25
THEN
    SommeR := r1 + r2
END
```

6.11 Substitution définition locale

Syntaxe

Substitution_définition_locale ::=
"LET" Ident$^{+","}$ "BE"
(Ident "=" *Expression*)$^{+"\wedge"}$
"IN" *Substitution* "END"

Définition

Soient $x1, ..., xn$ une liste non vide d'identificateurs deux à deux distincts avec $n \geq 1$, $E1, ..., En$ une liste d'expressions et S une substitution, alors :

LET $x1, ..., xn$ BE $x1 = E1 \wedge ... \wedge xn = En$ IN S END $\triangleq$
ANY $x1, ..., xn$ WHERE $x1 = E1 \wedge ... \wedge xn = En$ THEN S END

Restrictions

1. La substitution LET n'est pas une substitution d'implantation.
2. Les identificateurs introduits dans une substitution LET doivent être deux à deux distincts.
3. Chaque identificateur xi introduit dans une substitution LET L BE P IN S END doit être défini une et une seule fois à l'aide d'un prédicat de typage de donnée abstraite de la forme $xi = Ei$.
4. Seuls les identificateurs xi introduits après le mot réservé LET peuvent apparaître en partie gauche des prédicats introduits par le mot réservé BE.

Règle de portée

Dans une substitution LET L BE P IN S END, les identificateurs de la liste L sont accessibles en partie gauche des prédicats de typage qui composent le prédicat P, mais pas dans les expressions en partie droite et ils sont accessibles en lecture seulement dans la substitution S.

Description

La substitution LET L BE P IN S END introduit une liste de données abstraites L dont la valeur est donnée par le prédicat P et qui peuvent être utilisés en lecture dans la substitution S.

Le prédicat P est constitué d'une liste de conjonctions de la forme $x_i = E_i$ où x_i est un identificateur de la liste L et où E_i est une expression. Chaque identificateur x_i doit être défini une et une seule fois dans P, son expression associée E_i ne doit utiliser aucun des identificateurs de L. Pour définir des identificateurs qui dépendent d'autres identificateurs, il suffit d'utiliser deux substitutions LET imbriquées. Les identificateurs introduits par la substitution LET peuvent être utilisés en lecture seulement dans la substitution S.

Exemples

```
LET r1, r2 BE
    r1 = (Var1 + Var2) / 2 ∧
    r2 = (Var1 - Var2) / 2
IN
    SommeR := r1 + r2 ||
    DifferenceR := r1 - r2
END
```

6.12 Substitution devient élément de

Opérateur

$:\in$ Devient élément de

Syntaxe

Substitution_devient_elt_de ::= *Ident_ren*$^{+","}$ ":∈" *Expression*

Définition

Soient E un ensemble, X une liste de variables modifiables non vide. Soit Y une liste de variables intermédiaires ayant autant d'éléments que X mais *non libre* dans X et E. Alors :

$X :\in E \;\; \widehat{=} \;\;$ ANY Y WHERE $Y \in E$ THEN $X := Y$ END

Règle de typage

Dans la substitution $X :\in E$, si X est de type T alors E doit être de type $\mathbb{P}(T)$.

Restrictions

1. La substitution « devient élément de » n'est pas une substitution d'implantation.
2. Les identificateurs introduits dans une substitution « devient élément de » doivent être deux à deux distincts.
3. Les variables X d'une substitution $X :\in E$ doivent être des données modifiables.

Description

La substitution « devient élément de » permet de remplacer des variables par des valeurs appartenant à un certain ensemble. Les variables doivent être deux à deux distinctes. Si l'ensemble a plusieurs valeurs, la substitution définit alors un comportement non déterministe.

Exemples

```
i1 :∈ INT ;
b1 :∈ BOOL ;
x1 :∈ -10 .. 10 ;
y1, y2 :∈ {1, 3, 5} × NAT
```

6.13 Substitution devient tel que

Opérateur

:( ) Devient tel que

Syntaxe

Substitution_devient_tel_que ::= *Ident_ren*$^{+","}$ ":" "(" *Prédicat* ")"

Définition

Soient P un prédicat et X une liste de variables modifiables deux à deux distinctes. Soit Y une liste de variables intermédiaires ayant autant d'éléments que X, ne figurant pas dans X et *non libre* dans P, alors :

1. $X : (P) \triangleq$ ANY Y WHERE $[X := Y]\, P$ THEN $X := Y$ END

Soit une variable y de X. La notation $y\$0$ est utilisable au sein de P. Elle représente la valeur de la variable y avant l'application de la substitution « devient tel que », alors :

2. $X :(P) \triangleq$ ANY Y WHERE $[X, y\$0 := Y, y]P$ THEN $X := Y$ END

Restrictions

1. La substitution « devient tel que » n'est pas une substitution d'implantation.
2. Les variables X d'une substitution $X : (P)$ doivent être accessibles en écriture.
3. Dans l'expression $X : (P)$, les variables de la liste X, doivent être typées dans le prédicat P à l'aide de prédicats de typage de données abstraites situés dans une liste de conjonctions, au plus haut niveau d'analyse syntaxique de P.

Description

La substitution « devient tel que » permet de remplacer des variables par des valeurs qui satisfont un prédicat donné. Les variables doivent être deux à deux distinctes. Si plusieurs valeurs satisfont le prédicat, la substitution ne précise pas laquelle est effectivement choisie, son comportement est alors non déterministe.

La valeur avant substitution d'une variable y de X peut être référencée par $y\$0$ dans le prédicat P. Cette possibilité est une facilité d'écriture qui évite d'introduire une variable intermédiaire dans une substitution `ANY`.

Exemples

$x : (x \in \mathbb{Z} \wedge x > -4 \wedge x < 4)$;
$a, b : (a \in \mathrm{INT} \wedge b \in \mathrm{INT} \wedge a^2 + b^2 = 25)$;
$y : (y \in \mathrm{NAT} \wedge y\$0 > y)$

Cette dernière substitution aurait pu s'écrire sans utiliser la notation $0, de la manière suivante :

```
ANY y2 WHERE
    y2 ∈ NAT ∧ y > y2
THEN
    y := y2
END
```

6.14 Substitution variable locale

Syntaxe

Substitution_variable_locale ::= "VAR" Ident$^{+","}$ "IN" *Substitution* "END"

Définition

Soient X une liste de variables deux à deux distinctes, S une substitution et P un prédicat, alors :

$$[\text{VAR } X \text{ IN } S \text{ END}]\, P \;\Leftrightarrow\; \forall X \,.\, [S]\, P$$

Règle de portée

Dans une substitution VAR X IN S END, les identificateurs de la liste X sont accessibles en lecture et en écriture dans la substitution S.

Restrictions

1. La substitution VAR n'est pas une substitution de machine abstraite.
2. Les variables introduites par une substitution VAR doivent être deux à deux distinctes.
3. En implantation, les variables locales doivent être initialisées avant d'être lues. Pour qu'une instruction conditionnelle ou une instruction cas initialise une variable locale, chaque branche de l'instruction doit initialiser la variable locale.

Description

La substitution VAR L IN S END introduit une liste non vide de variables locales L deux à deux distinctes. Ces variables locales sont utilisables dans la substitution S. Elles sont typées lors de leur première utilisation dans l'ordre d'écriture des substitutions constituant S. Les mots réservés IN et END parenthèsent la substitution S comme une substitution de bloc BEGIN END.

Exemples

```
VAR  varLoc1, varLoc2  IN
      varLoc1   :=   x1 + 1   ;
      varLoc2   :=   2 × varLoc1   ;
      x1   :=   varLoc2
END
```

6.15 Substitution séquencement

Opérateur

; Séquencement

Syntaxe

Substitution_séquence ::= *Substitution* ";" *Substitution*

Définition

Soient S et T des substitutions et P un prédicat, alors :

$$[S\ ;\ T]\ P \ \Leftrightarrow\ [S][T]\ P$$

Ce qui signifie que le prédicat obtenu par application de la substitution séquence S ; T sur le prédicat P est le prédicat obtenu par application de S sur le résultat de l'application de T sur P.

Restriction

1. La substitution séquencement n'est pas une substitution de machine abstraite.

Description

La substitution séquencement correspond à l'application en séquence de deux substitutions. L'application à un prédicat des substitutions regroupées dans un séquencement de substitutions se déroule dans l'ordre inverse du séquencement.

Ce résultat se retrouve intuitivement en considérant l'exemple suivant : $[x := 0\ ;\ x := 1]\ P$. D'après la définition de la substitution séquencement, on commence par remplacer toutes les occurrences de x dans P par 1, puis on remplace les occurrences de x (qui ont disparu) dans le prédicat obtenu par 0, ce qui est sans effet. Considérons maintenant seulement la substitution de séquencement, sa signification opérationnelle est que x devient d'abord égal à 0 puis ensuite à 1, ainsi la valeur finale de x est bien 1, donc en appliquant la substitution séquencement à P, on obtient le même résultat que précédemment.

Exemples

z := *x* ; *x* := *y* ; *y* := *z*

L'exemple précédent échange les valeurs des variables x et y.

6.16 Substitution appel d'opération

Opérateur

$\longleftarrow$ Appel d'opération

Syntaxe

Substitution_appel_opération ::= [*Ident_ren*$^{+","}$ "$\longleftarrow$"] *Ident_ren* ["(" *Expression*$^{+","}$ ")"]

Définitions

1. Soit op une opération (non locale ou locale) sans paramètre de sortie et sans paramètre d'entrée, définie par op = S, alors la signification d'un appel de op est :

   [op] P $\Leftrightarrow$ $[S]\,P$

2. Soit op une opération (non locale ou locale) sans paramètre de sortie et avec des paramètres d'entrée, définie par op (X) = S où X est une liste d'identificateurs désignant les paramètres formels d'entrée de op, et soit E une liste d'expressions représentant les paramètres d'entrée effectifs de op, alors la signification d'un appel de op (E) est :

   [op (E)] P $\Leftrightarrow$ $[X := E\ ;\ S]\,P$

3. Soit op une opération (non locale ou locale) avec des paramètres de sortie et sans paramètre d'entrée, définie par $Y \longleftarrow$ op = S, où Y est une liste d'identificateurs désignant les paramètres formels de sortie de op, et soit R une liste d'identificateurs éventuellement renommés désignant les paramètres effectifs de sortie de op, alors la signification d'un appel de $R \longleftarrow$ op est :

   [$R \longleftarrow$ op] P $\Leftrightarrow$ $[S\ ;\ R := Y]\,P$

4. Si op est une opération (non locale ou locale) avec des paramètres de sortie et des paramètres d'entrée, définie par $Y \longleftarrow$ op $(X) = S$, la signification d'un appel de $R \longleftarrow$ op (E) est :

   [$R \longleftarrow$ op (E)]P $\Leftrightarrow$ $[X := E\ ;\ S\ ;\ R := Y]\,P$

Règles de typage

Dans les appels d'opération op (E) et $R \longleftarrow$ op (E), E est une liste d'expressions dont le type doit être identique au type des paramètres d'entrée de l'opération op, défini dans la machine abstraite dans laquelle cette opération est déclarée.

Dans les appels d'opération $R \longleftarrow$ op et $R \longleftarrow$ op (E), R est une liste de noms de données dont le type doit être identique au type des paramètres de sortie de l'opération op, défini dans la spécification dans laquelle cette opération est déclarée.

Restriction

1. Une variable ne doit pas être utilisée plusieurs fois comme paramètre de retour effectif d'un appel d'opération. Par exemple, l'appel d'opération $x, x \longleftarrow$ op est interdit.

Description

La substitution appel d'opération permet d'appliquer la substitution d'une opération (non locale ou locale), en remplaçant les paramètres formels par des paramètres effectifs. Les paramètres d'entrée éventuels sont des expressions et les paramètres de sortie éventuels sont des données accessibles en écriture.

L'appel d'opération se définit sous quatre formes différentes, selon la présence de paramètres d'entrée et de sortie.

Exemples

opa ;
opb (x + 1, TRUE) ;
res1, *res2* ⟵ opc ;
res, *flag* ⟵ opd (x)

6.17 Substitution boucle tant que

Syntaxe

Substitution_tant_que ::=
"WHILE" *Condition* "DO" *Instruction*
"INVARIANT" *Prédicat*
"VARIANT" *Expression*
"END"

Définition

Soient P un prédicat, S une substitution, I et R des prédicats et V une expression. Si X représente la liste des variables libres apparaissant dans S et I et n une variable fraîche, c'est-à-dire non libre dans V, I, P et S, alors :

$$
\begin{array}{l}
[\mathrm{WHILE}\ P\ \mathrm{DO}\ S\ \mathrm{INVARIANT}\ I\ \mathrm{VARIANT}\ V\ \mathrm{END}]\ R \quad \Leftrightarrow \\
\quad I \ \wedge \\
\quad \forall X . (I \wedge P \ \Rightarrow \ [S]\, I) \ \wedge \\
\quad \forall X . (I \Rightarrow V \in \mathbb{N}) \ \wedge \\
\quad \forall X . (I \wedge P \ \Rightarrow \ [n := V ; S](\mathrm{V} < \mathrm{n})) \ \wedge \\
\quad \forall X . (I \wedge \neg P \ \Rightarrow \ R))
\end{array}
$$

Règle de typage

Dans une substitution WHILE P DO S INVARIANT I VARIANT V END, le variant V doit être de type $\mathbb{Z}$.

Restriction

1. La substitution « tant que » n'est pas une substitution de machine abstraite ni de raffinement.

Description

La substitution « tant que » réalise une boucle « tant que ». La substitution WHILE P DO S INVARIANT I VARIANT V END réalise la substitution S tant que le prédicat P reste vrai. Une boucle « tant que » doit se terminer au bout d'un nombre fini d'itérations.

I est l'invariant de la boucle. C'est un prédicat donnant des propriétés sur les variables utilisées dans la boucle. L'invariant de boucle permet de prouver qu'à chaque pas la boucle est possible et qu'elle donne bien le résultat produit à la sortie.

Dans le cas où une variable y de l'abstraction de l'implantation est collée par homonymie avec une variable concrète de l'implantation ou bien avec une variable d'une machine *importée*, alors il est possible d'utiliser la notation $y\$0$ dans l'invariant de la boucle. La notation $y\$0$ désigne la valeur de y au début de l'opération de l'implantation, alors que y désigne la valeur courante de la variable de l'implantation ou de la machine *importée*.

V est le variant de la boucle. C'est une expression entière qui permet de démontrer que la « boucle tant que » se termine au bout d'un nombre fini d'itérations. Pour cela, il faut prouver que V est une expression entière, positive qui décroît strictement à chaque itération.

Dans le cas le plus général, la substitution WHILE peut être précédée, en séquence, d'une substitution qui initialise les variables utilisées dans la boucle. Elle prend alors la forme : *T* ; WHILE *P* DO *S* INVARIANT *I* VARIANT *V* END où *T* est la substitution d'initialisation.

Exemples

```
BEGIN
    varLoc := var1 ;
    cpt := 0
END ;
WHILE cpt < 5 DO
    varLoc := varLoc + 1 ;
    cpt := cpt + 1
INVARIANT
    cpt ∈ NAT ∧
    cpt ≤ 5 ∧
    varLoc ∈ NAT ∧
    varLoc = var1 + cpt
VARIANT
    5 - cpt
END
```

6.18 Substitution simultanée

Opérateur

|| Substitution simultanée

Syntaxe

Substitution_simultanée ::= *Substitution* "||" *Substitution*

Définition

La substitution simultanée se définit de manière inductive à partir de certaines propriétés et par rapport aux autres substitutions du langage. Soient $S1$, $S2$, $S3$, T et U des substitutions, x et y des variables distinctes, X une liste d'identificateurs, E une expression, $l1$ et $l2$ des listes de constantes, $P1$, $P2$ et R des prédicats, alors :

Propriétés de la substitution simultanée :

1. $S1 \parallel S2 \;=\; S2 \parallel S1$
2. $S1 \parallel (S2 \parallel S3) \;=\; (S1 \parallel S2) \parallel S3$

Définition de la substitution « devient égal » simultanée :

3. $x := E \;\parallel\; y := F \quad=\quad x, y := E, F$
4. $[x := E \;\parallel\; x := F]\,R \quad\Leftrightarrow\quad \forall x' . \,((x' = E \wedge x' = F) \;\Rightarrow\; [x := x']\,R)$

Définition de la substitution simultanée par rapport aux autres substitutions :

5. skip || S = S
6. X :(P) || S = ANY Y WHERE [X := Y]P THEN X := Y || S END
7. X :∈ E || S = ANY Y WHERE Y ∈ E THEN X := Y || S END
8. CHOICE S OR T END || U = CHOICE S || U OR T || U END
9. PRE P THEN S END || T = PRE P THEN S || T END
10. ASSERT P THEN S END || T = ASSERT P THEN S || T END
11. BEGIN S END || T = BEGIN S || T END
12. Si aucune variable élémentaire de X n'est libre dans T, alors :
    ANY X WHERE P THEN S END || T = ANY X WHERE P THEN S || T END
13. Si aucune variable élémentaire de X n'est libre dans T, alors :
    SELECT X THEN $S1$ WHEN $P2$ THEN $S2$ END || T = SELECT $P1$ THEN $S1$ || T WHEN $P2$ THEN $S2$ || T END
14. Si aucune variable élémentaire de X n'est libre dans T, alors :
    LET X BE $P1$ IN S1 END || T = LET X BE $P1$ IN $S1$ || T END
15. IF $P1$ THEN $S1$ ELSE $S2$ END || T = IF $P1$ THEN $S1$ || T ELSE $S2$ || T END
16. CASE E OF EITHER $l1$ THEN $S1$ OR $l2$ THEN $S2$ END =
    CASE E OF EITHER $l1$ THEN $S1$ || T OR $l2$ THEN $S2$ || T END
17. Si aucune variable élémentaire de X n'est libre dans T, alors :
    VAR X THEN S END || T = VAR X THEN S || T END

Restriction

1. La substitution simultanée n'est pas une substitution d'implantation.
2. Soient S_X et T_Y des substitutions qui modifient les listes de variables X et Y (cette

modification peut avoir lieu à un niveau d'imbrication quelconque dans les substitutions S_X et T_Y, en particulier dans le corps d'un appel d'opération ; chaque variable de X et Y apparaissant finalement en partie gauche d'une substitution « devient égal », « devient élément de », « devient tel que » ou comme paramètre de sortie d'un appel d'opération). Alors, il faut que les listes de variables X et Y soient disjointes.

3. Il est interdit d'appeler simultanément deux opérations d'écriture d'une même instance de machine *incluse*. En effet, même si chaque appel d'opération préserve l'invariant de l'instance de la machine *incluse*, il se peut que l'appel simultané de deux opérations brise cet invariant (cf. [B-Book] §7.2.3).
4. Dans une spécification d'opération locale (cf. §7.24 *La clause LOCAL_OPERATIONS*), il est interdit d'appeler simultanément deux opérations d'une même instance de machine *importée*.

Description

La substitution simultanée correspond à l'exécution simultanée de deux substitutions. Le caractère de simultanéité dénote le fait que les substitutions doivent pouvoir se réaliser indépendamment l'une de l'autre. La substitution simultanée est commutative et associative. La substitution simultanée de deux substitutions S et T modifiant des variables différentes, se définit par la conjonction des préconditions et la conjonction des postconditions de S et T (cf. [B-Book] §7.1.1). Pour rester cohérent avec la présentation des autres substitutions donnée dans ce chapitre, nous donnons ici une définition par rapport aux autres substitutions du langage.

Exemple

$x := y \;\|$
$y := x$

Dans l'exemple ci-dessus, les valeurs des variables x et y sont échangées.

7 COMPOSANTS

7.1 Machine abstraite

Syntaxe

Machine_abstraite ::=
"MACHINE" *En-tête*
*Clause_machine_abstraite**
"END"

Clause_machine_abstraite ::=
Clause_constraints
| *Clause_sees*
| *Clause_includes*
| *Clause_promotes*
| *Clause_extends*
| *Clause_uses*
| *Clause_sets*
| *Clause_concrete_constants*
| *Clause_abstract_constants*
| *Clause_properties*
| *Clause_concrete_variables*
| *Clause_abstract_variables*
| *Clause_invariant*
| *Clause_assertions*
| *Clause_initialisation*
| *Clause_operations*

Restriction

1. Une clause ne doit pas apparaître plus d'une fois dans une machine abstraite.

Description

Une machine abstraite est un composant qui définit à l'aide de différentes clauses, des données et leurs propriétés ainsi que des opérations. Une machine abstraite constitue la spécification d'un module B. Elle se compose d'un en-tête et d'un certain nombre de clauses. L'ordre des clauses dans un composant n'est pas imposé. La description des clauses est donnée par le tableau ci-dessous.

Clause	**Description**	**Réf.**
CONSTRAINTS	définition du type et des propriétés des paramètres scalaires formels	§7.5
SEES	liste des instances de machines *vues*	§7.8
INCLUDES	liste des instances de machines *incluses*	§7.9
PROMOTES	liste des opérations promues des instances de machines *incluses*	§7.10
EXTENDS	liste des instances de machines *étendues*	§7.11
USES	liste des instances de machines *utilisées*	§7.12
SETS	liste des ensembles abstraits et définition des ensembles énumérés	§7.13
CONCRETE_CONSTANTS	liste des constantes concrètes	§7.14
ABSTRACT_CONSTANTS	liste des constantes abstraites	§7.15

Clause	**Description**	**Réf.**
PROPERTIES	définition du type et des propriétés des constantes de la machine	§7.16
CONCRETE_VARIABLES	liste des variables concrètes	§7.18
ABSTRACT_VARIABLES	liste des variables abstraites	§7.19
INVARIANT	définition du type et des propriétés des variables	§7.20
ASSERTIONS	définition de propriétés déductibles de l'invariant	§7.21
INITIALISATION	initialisation des variables	§7.22
OPERATIONS	liste et définition des opérations propres	§7.23

Utilisation

Table de visibilité

Soit M_A une machine abstraite. La table de visibilité suivante précise le mode d'accès de chaque constituant de M_A (donnée ou opération), dans les clauses de M_A.

Par exemple, nous voyons qu'une variable concrète peut être lue dans les clauses INVARIANT et ASSERTIONS et peut être lue et écrite, c'est-à-dire modifiée, dans la clause INITIALISATION ou dans une opération.

On appelle opération propre d'un composant, une opération dont le corps est défini dans le composant, au sein de la clause OPERATIONS. Les autres opérations du composant sont les opérations promues par le composant (cf. §7.10 *La clause PROMOTES*).

Clauses de M_A / Constituants de M_A	CONSTRAINTS	Paramètres d'INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS
Paramètres formels	visible	visible		visible	visible
Ensembles, énumérés littéraux, constantes concrètes		visible	visible	visible	visible
Constantes abstraites, non homonymes		visible	visible	visible	visible
Variables concrètes, non homonymes				visible	visible - modifiable
Variables abstraites, non homonymes				visible	visible - modifiable
Opérations propres (non promues)					

7.2 En-tête de composant

Syntaxe

En-tête ::= Ident ["(" $\text{Ident}^{+","}$ ")"]

Restrictions

1. Le nom d'un composant doit être unique dans le projet.
2. Les déclarations des paramètres formels d'un raffinement ou d'une implantation doivent être syntaxiquement identiques aux déclarations que l'on trouve dans la machine abstraite raffinée par ces composants.
3. Le nom des paramètres ensembles ne doit pas comporter de caractère minuscule.
4. Le nom des paramètres scalaires doit comporter au moins un caractère minuscule.

Description

L'en-tête d'un composant définit le nom du composant et la liste de ses paramètres.

Les paramètres formels d'un composant sont recopiés dans l'en-tête des différents raffinements du composant ainsi que dans son implantation. Ils paramètrent les instances de la machine abstraite. Il peut s'agir soit d'ensembles abstraits servant de base au typage (cf. §7.13 *La clause SETS*), soit de scalaires (cf. §7.5 *La clause CONSTRAINTS*).

Utilisation

Les paramètres formels sont visibles dans les clauses INVARIANT et ASSERTIONS, INCLUDES (comme paramètre effectif de machine abstraite *incluse*), ainsi que dans les clauses INITIALISATION et OPERATIONS.

On notera que les paramètres formels ne sont pas visibles dans la clause PROPERTIES.

7.3 Raffinement

Syntaxe

Raffinement ::=
    "REFINEMENT" *En-tête*
    Clause_refines
    *Clause_raffinement**
    "END"

Clause_raffinement ::=
    Clause_sees
  | *Clause_includes*
  | *Clause_promotes*
  | *Clause_extends*
  | *Clause_sets*
  | *Clause_concrete_constants*
  | *Clause_abstract_constants*
  | *Clause_properties*
  | *Clause_concrete_variables*
  | *Clause_abstract_variables*
  | *Clause_invariant*
  | *Clause_assertions*
  | *Clause_initialisation*
  | *Clause_operations*

Description

Un raffinement est un composant qui raffine une machine abstraite ou un autre raffinement (cf. §8.2 *Module B*).

Les clauses CONSTRAINTS et USES sont interdites dans un raffinement.

Restriction

1. Une clause ne doit pas apparaître plus d'une fois dans un raffinement.

Utilisation

Un raffinement peut être référencé par son nom dans la clause REFINES d'un autre raffinement, pour déclarer que ce dernier raffine le premier.

Tables de visibilité

Soit M_N un raffinement. La table de visibilité suivante précise le mode d'accès de chaque constituant de M_N, dans les clauses de M_N.

Clauses de M_N / Constituants de M_N	Paramètres d'INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS
Paramètres formels	visible		visible	visible
Ensembles, énumérés littéraux, constantes concrètes, non homonymes	visible	visible	visible	visible
Constantes abstraites non homonymes	visible	visible	visible	visible
Variables concrètes non homonymes			visible	visible - modifiable
Variables abstraites non homonymes			visible	visible - modifiable
Opérations propres (non promues)				

Soit M_{N-1} l'abstraction de M_N, c'est-à-dire le composant raffiné par M_N. La table de visibilité suivante précise le mode d'accès de chaque constituant de M_{N-1} disparaissant dans M_N, dans les clauses de M_N. On ne s'intéresse ici qu'aux données abstraites de M_{N-1} car les autres données sont conservées dans M_N.

Dans le cas de l'initialisation et des opérations, on distingue la visibilité dans les prédicats des substitutions assertions (cf. §6.5 *Substitution assertion*) et dans le reste des substitutions.

Clauses de M_N / Constituants de M_{N-1}	Paramètres d'INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS	
				Substitutions	Prédicats d'ASSERT
Constantes abstraites disparaissant dans M_N		visible	visible		visible
Variables abstraites disparaissant dans M_N			visible		visible

7.4 Implantation

Syntaxe

Implantation ::=
"IMPLEMENTATION" *En-tête*
Clause_refines
*Clause_implantation**
"END"

Clause_implantation ::=
Clause_sees
| *Clause_imports*
| *Clause_promotes*
| *Clause_extends_B0*
| *Clause_sets*
| *Clause_concrete_constants*
| *Clause_properties*
| *Clause_values*
| *Clause_concrete_variables*
| *Clause_invariant*
| *Clause_assertions*
| *Clause_initialisation_B0*
| *Clause_operations_B0*

Restriction

1. Une clause ne peut apparaître qu'une seule fois au plus dans une implantation.

Description

Une implantation est un composant qui constitue le dernier raffinement d'une machine abstraite (cf. §8.2 *Module B*).

Deux nouvelles clauses peuvent apparaître dans une implantation : la clause IMPORTS et la clause VALUES. La clause IMPORTS crée des instances concrètes de machines abstraites dans un projet. Les opérations de ces machines *importées* sont appelées dans les opérations de l'implantation. La clause VALUES permet de donner une valeur aux ensembles abstraits et aux constantes concrètes de l'implantation.

La clause EXTENDS, correspond à la clause IMPORTS dans une implantation alors qu'elle correspond à la clause INCLUDES dans une machine abstraite ou un raffinement.

Les clauses INITIALISATION et OPERATIONS diffèrent dans une implantation et dans une machine abstraite ou un raffinement. Dans une implantation, ces clauses sont constituées d'expressions ou de substitutions concrètes (cf. §7.24 *La clause LOCAL_OPERATIONS*).

Les clauses CONSTRAINTS, INCLUDES, USES, ABSTRACT_CONSTANTS et ABSTRACT_VARIABLES (ou VARIABLES) sont interdites dans une implantation.

Utilisation

Tables de visibilité

Soit M_N une implantation. La table de visibilité suivante précise le mode d'accès de chaque constituant de M_N, dans les clauses de M_N. Dans le cas de l'initialisation et des opérations, on distingue l'utilisation des constituants dans les parties non traduites (le variant et l'invariant de « boucle tant que » et le prédicat d'une assertion), et dans les parties traduites (le reste des instructions). Cette table ne donne pas les règles de visibilité des constantes concrètes de M_N homonymes avec des constantes concrètes de machines *vues* ou *importées* ni des variables concrètes de M_N homonymes avec des variables concrètes d'instances de machines *importées*. En effet, en cas d'implantation par homonymie de constantes ou de variables concrètes, celles-ci suivent les règles de visibilité des machines *vues* (cf. §7.8 *La clause SEES*) ou *importées* (cf. §7.7 *La clause IMPORTS*).

Clauses de M_N / Constituants de M_N	Paramètres d'IMPORTS / EXTENDS	PROPERTIES	VALUES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS: Instructions	INITIALISATION / OPERATIONS: variants et invariants de boucles, prédicats d'ASSERT	LOCAL_ OPERATIONS
Paramètres formels	visible			visible	visible	visible	visible
Ensembles énumérés, énumérés littéraux, non homonymes	visible	visible	visible	visible	visible	visible	visible
Ensembles abstraits, constantes concrètes, non homonymes	visible	visible	visible modifiable	visible	visible	visible	visible
Variables concrètes non homonymes				visible	visible – modifiable	visible	visible - modifiable
Opérations propres (non promues)							
Opérations locales					visible – modifiable dans OPERATIONS, pas dans INITIALISATION		

Soit M_{N-1} l'abstraction de M_N. La table de visibilité suivante indique le mode d'accès de chaque constituant de M_{N-1} disparaissant dans M_N, dans les clauses de M_N.

Clauses de M_N / Constituants de M_{N-1}	Paramètres d'IMPORTS / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS: Instructions	INITIALISATION / OPERATIONS: Variants et invariants de boucles, prédicats d'ASSERT	LOCAL_OPERATIONS: Substitutions	LOCAL_OPERATIONS: Prédicats d'ASSERT
Constantes abstraites		visible	visible		visible		visible
Variables abstraites			visible		visible		visible

7.5 La clause CONSTRAINTS

Syntaxe

Clause_constraints ::= "CONSTRAINTS" *Prédicat*

Restrictions

1. Chaque paramètre scalaire de la machine abstraite doit être typé par un prédicat de typage des paramètres de machines (cf. §3.8 *Typage des paramètres de machines*) situé au premier niveau d'une liste de conjonctions. Les paramètres scalaires ne peuvent pas être utilisés dans la clause CONSTRAINTS avant d'avoir été typés.

Description

La clause CONSTRAINTS permet de typer les paramètres scalaires de la machine abstraite et d'exprimer des propriétés complémentaires, encore appelées contraintes, portant sur ces paramètres.

Les paramètres de machine abstraite sont de deux sortes :

- les paramètres scalaires : le nom d'un paramètre scalaire est un identificateur qui doit contenir au moins un caractère minuscule. Son type peut être $\mathbb{Z}$, BOOL ou un paramètre ensemble de la machine (voir ci-dessous),
- les paramètres ensembles : le nom d'un paramètre ensemble est un identificateur qui ne doit pas contenir de caractère minuscule. Les paramètres ensembles sont des types de base (cf. §3.2 *Les types B*). Ils peuvent servir à typer certains des paramètres scalaires de la machine.

Utilisation

Les paramètres formels ensembles définissent de nouveaux types, de manière similaire aux ensembles abstraits et aux ensembles énumérés (cf. §7.13 *La clause SETS*). Le contrôle de type interdit par conséquent d'exprimer dans la clause CONSTRAINTS qu'un paramètre ensemble est égal à un intervalle d'entiers, ou bien qu'un paramètre ensemble est contenu dans un autre, puisque les types en parties gauche et droite sont incompatibles.

Exemple

Dans l'exemple ci-dessous, *p1* est un paramètre entier, *p2* est un paramètre booléen, *p3* est un paramètre appartenant au paramètre ensemble *ENS1*. Par ailleurs *ENS1* et *ENS2* sont des paramètres ensembles.

```
MACHINE
    MA ( p1, p2, p3, ENS1, ENS2 )
CONSTRAINTS
    p1 ∈ INT  ∧
    p2 ∈ BOOL  ∧
    p3 ∈ ENS1  ∧
    card (ENS1) = 10
...
END
```

7.6 La clause REFINES

Syntaxe

Clause_refines ::= "REFINES" Ident

Restriction

1. La clause REFINES doit contenir le nom d'une machine abstraite ou d'un raffinement.

Description

La clause REFINES d'un raffinement contient le nom du composant raffiné par ce raffinement, c'est-à-dire son abstraction.

7.7 La clause IMPORTS

Syntaxe

Clause_IMPORTS ::=
 "IMPORTS" (*Ident_ren* ["(" *Instanciation_B0*$^{+","}$ ")"])$^{+","}$

Instanciation_B0 :=
 Terme
| *Ensemble_nombres_B0* (cf. §3.5)
| "BOOL"
| *Intervalle_B0* (cf. §5.7)

Règle de typage

- Dans le cas où une instance de machine *importée* possède des paramètres, les paramètres effectifs doivent respecter la règle suivante. Les paramètres effectifs correspondant à des données scalaires doivent être de même type que les paramètres formels de la machine *importée* tels qu'ils sont stipulés dans la clause CONSTRAINTS de cette machine. Les paramètres effectifs correspondant à des ensembles doivent être de type $\mathbb{P}(T)$ où T est un type de base autre que STRING.

Restrictions

1. Les identificateurs d'une clause IMPORTS doivent désigner des machines abstraites.
2. Les identificateurs renommés ne peuvent avoir au plus qu'un préfixe de renommage.
3. Chaque nom de machine doit être suivi d'une liste de paramètres effectifs de même nombre que les paramètres de la machine *importée*.
4. Si une constante concrète d'une instance de machine *importée* par une implantation est homonyme à une constante concrète de l'implantation, alors les deux constantes homonymes désignent la même donnée et elles doivent donc être de même type.
5. Si une variable concrète d'une instance de machine *importée* par une implantation est homonyme à une variable concrète de l'implantation, alors les deux variables homonymes désignent la même donnée et elles doivent être de même type.
6. Une instance de machine ne doit pas être *importée* plusieurs fois dans un projet (cf. §8.3 Règle n°1 sur les liens IMPORTS).
7. Tout projet complet doit contenir un et un seul module qui n'est jamais instancié par *importation* dans le projet (cf. §8.3 Règle n°2 sur les liens IMPORTS).
8. Un composant ne peut pas posséder plusieurs liens sur une même instance de machine. Par exemple, une implantation ne peut pas *voir* (cf. §7.8 *Clause SEES*) et *importer* une même instance de machine (cf. §8.3 Règle n°6 sur les liens de dépendance).
9. Il ne doit pas exister de cycle dans le graphe de dépendance d'un projet (cf. §8.3 Règle n°7 sur les liens de dépendance).

Description

Le lien d'*importation* entre une implantation et une instance de machine abstraite est un lien de composition. Il permet de réaliser l'implantation du module concerné sur les instances de machines *importées*. L'implantation crée l'instance de machine abstraite *importée* afin de se servir de ses données et de ses opérations pour implémenter ses propres données et opérations. Le module de l'implantation est donc le père de l'instance de module *importé* dans le graphe d'*importation* du projet (cf. §8.3 *lien IMPORTS*). Cette implantation est la seule implantation du projet qui possède le droit de modifier les variables de l'instance de machine *importée*, par l'intermédiaire des opérations adéquates de la machine *importée*.

Utilisation

La clause IMPORTS contient la déclaration de la liste des instances de machines *importées*. Il s'agit soit du nom de la machine seul, qui référence l'instance sans renommage de la machine (l'instance de la machine est alors confondue avec celle-ci), soit du nom de la machine précédé d'un renommage, qui désigne l'instance de la machine renommée (dans ce cas on peut avoir plusieurs instances de la machine, chaque instance correspondant à un préfixe distinct). Si une instance de machine *importée* est paramétrée, les paramètres effectifs de la machine doivent être fournis, ce qui permet d'instancier les paramètres formels de la nouvelle instance de machine. Les paramètres d'une machine abstraite sont décrits dans la clause CONSTRAINTS (cf. §7.5 *La clause CONSTRAINTS*). Ce sont soit des scalaires, soit des ensembles de scalaires. Une Obligation de Preuve est générée afin de prouver que les paramètres effectifs d'une instance de machine *importée* respectent les contraintes de la machine.

Instanciation des paramètres scalaires

Un paramètre effectif scalaire d'une instance de machine importée peut être :

- un booléen littéral TRUE ou FALSE, ou une instruction booléenne,
- un élément d'un ensemble énuméré de l'implantation ou d'une instance de machine *vue* (cf. §7.8 *La clause SEES*),
- un paramètre formel scalaire de l'implantation,
- une constante concrète de l'implantation ou d'une instance de machine *vue*, appartenant à INT, à BOOL ou à un ensemble abstrait ou énuméré,
- une expression arithmétique formée de paramètres formels de l'implantation, de constantes concrètes de l'implantation ou d'instances de machines *vues*, et d'entiers littéraux. Les opérateurs arithmétiques permis sont '+', '-', '×', '/', mod, a^b, succ et pred. Une Obligation de Preuve est générée afin de prouver que chaque sous-expression arithmétique est bien définie et que son résultat appartient à l'ensemble des entiers concrets INT (cf. §7.25.2 *Les termes*).

Instanciation des paramètres ensembles

L'instanciation d'un paramètre ensemble d'une instance de machine *importée* peut être :

- un paramètre formel ensemble de l'implantation,
- un ensemble abstrait ou énuméré de l'implantation ou d'une instance de machine *vue*,
- une constante concrète de l'implantation ou d'une instance de machine *vue*, incluse dans $\mathbb{Z}$ ou dans un ensemble abstrait,
- un intervalle non vide de INT dont les bornes sont formées d'expressions arithmétiques similaires à celles permises pour instancier les paramètres scalaires.

Exemple

```
MACHINE
    MA (E0 )
CONCRETE_CONSTANTS
    c1
PROPERTIES
    c1 ∈ 0 .. 10
...
END
```

```
MACHINE
    Msees
SETS
    COUL = { Rouge, Vert, Bleu }
CONCRETE _CONSTANTS
    c2
PROPERTIES
    c2 ∈ INT
END
```

```
IMPLEMENTATION
    MA_i (E0 )
REFINES
    MA
SEES
    Msees
IMPORTS
    Mimports ( E0, COUL, c1 .. (c2 + 2))
VALUES
    c1 = 2
...
END
```

```
MACHINE
    Mimports (ENS1, ENS2, ENS3)

...
END
```

Visibilité

Les paramètres formels des machines *importées* ne sont pas accessibles dans le composant qui *importe*. Les ensembles et les constantes concrètes sont accessibles dans les clauses PROPERTIES, VALUES, INVARIANT, ASSERTIONS et dans le corps de l'initialisation et des opérations de la machine. Les constantes abstraites sont accessibles dans les clauses PROPERTIES, INVARIANT, ASSERTIONS et dans les variants et invariants de boucle et dans le prédicat des instructions ASSERT des opérations et de l'initialisation. Les variables sont accessibles en lecture dans les invariants et les assertions. Les variables concrètes sont, en outre, accessibles en lecture dans le corps de l'initialisation et des opérations. Les variables abstraites sont accessibles uniquement dans les variants et invariants de boucle et dans le prédicat des instructions ASSERT des opérations et de l'initialisation. Il est possible d'utiliser les opérations d'une machine *importée* dans l'initialisation et dans les opérations de l'implantation.

Promotion des opérations

Les opérations d'une machine *importée* peuvent devenir automatiquement des opérations du composant qui réalise l'*importation*. Il s'agit du mécanisme de promotion d'opération (cf. §7.10 *La clause PROMOTES* et §7.11 *La clause EXTENDS*).

Table de visibilité

Soit M_N une implantation qui *importe* une instance de machine M_B. La table de visibilité suivante précise le mode d'accès de chaque constituant de M_B, dans les clauses de M_N.

Clauses de M_N / Constituants de M_B	Paramètres d'IMPORTS / EXTENDS	PROPERTIES	VALUES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS: Instructions	INITIALISATION / OPERATIONS: Variants et invariants de boucles, prédicats d'ASSERT	LOCAL_ OPERATIONS
Paramètres formels							
Ensembles, énumérés littéraux, constantes concrètes		visible	visible	visible	visible	visible	visible
Constantes abstraites		visible		visible		visible	visible
Variables concrètes				visible	visible – non modifiable	visible	visible - modifiable
Variables abstraites				visible		visible	visible - modifiable
Opérations					visible – modifiable		visible - modifiable

7.8 La clause SEES

Syntaxe

Clause_sees ::= "SEES" *Ident_ren*$^{+","}$

Restrictions

1. Les identificateurs d'une clause SEES (sans tenir compte des éventuels préfixes de renommage) doivent désigner des machines abstraites.
2. Si un composant C_A appartenant au *module* M_A *voit* une instance de machine $r.M_B$, et si le premier ancêtre commun dans l'arbre d'*importation* du projet de M_A et M_B est M_O, alors les renommages r (éventuellement vide) doivent provenir des renommages successifs dans l'arbre d'*importation* du projet subis depuis l'instance de module M_O jusqu'à l'instance de module M_B (cf. paragraphe ci-dessous *SEES et renommage*).
3. Si une instance de machine est *vue* par un composant d'un *module développé*, alors les raffinements de ce composant doivent également *voir* cette instance (cf. §8.3 Règle n°4 sur les liens SEES).
4. Un composant d'un module M_A ne peut pas *voir* une instance de module M_B *importée* par une instance de module dont M_A dépend (cf. §8.3 Règle n°5 sur les liens SEES).
5. Un composant ne peut pas posséder plusieurs liens sur une même instance de machine. Par exemple, une implantation ne peut pas *voir* et *importer* une même instance de machine (cf. §8.3 Règle n°6 sur les liens de dépendance).
6. Il ne doit pas exister de cycle dans le graphe de dépendance d'un projet (cf. §8.3 Règle n°7 sur les liens de dépendance).
7. Si un composant A *voit* une machine B, il faut pouvoir aller de A vers B en traversant l'arbre d'importation suivant la règle suivante : il faut partir depuis A, ensuite remonter au moins une fois et finalement descendre exactement une fois pour arriver sur B. Autrement dit on ne peut voir que ses frères, oncles, grand-oncles, grand-grand-oncles etc. (cf. §8.3 Règle n°8 sur les liens SEES).

Description

Le lien SEES permet de référencer dans un composant une instance de machine abstraite *importée* dans une autre branche du projet, afin de consulter ses constituants (ensembles, constantes et variables) sans les modifier.

Utilisation

La liste des instances de machines *vues* est constituée de noms de machines éventuellement renommées. La signification de ce renommage est donnée dans le paragraphe suivant. Si une machine est paramétrée, les paramètres effectifs de la machine ne doivent pas être fournis. En effet, ceux-ci sont uniquement fournis lors de la création d'instances de machines. Des instances de machines sont créées soit localement par *inclusion* (cf. §7.9 *La clause INCLUDES*), soit globalement par *importation* (cf. §7.7 *La clause IMPORTS*). La clause SEES ne fait que référencer une instance de machine globale.

SEES et renommage

Lorsqu'une machine M_A *voit* une instance de machine M_B, le nom de l'instance effectivement *vue* se construit à partir du nom de la machine *vue*, précédé des éventuels renommages successifs de l'instance de M_B, subis dans son arbre d'*importation*, à partir du premier ancêtre commun de M_A et M_B.

Considérons l'exemple suivant :

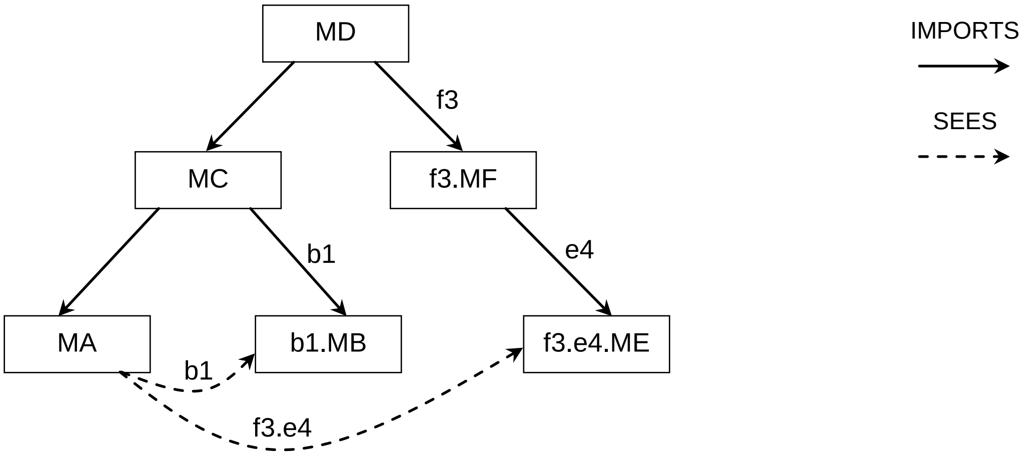


Figure 1 : exemple n°1 de SEES et renommage

Le schéma précédent représente le graphe de dépendance entre instances de machines d'un projet. *MA voit* les instances *b1.MB* et *f3.e4.MD*. En effet, *MC* est le premier ancêtre commun de *MA* et de *MB*. À partir de *MC*, *MB* est *importée* avec comme préfixe de renommage *b1*. De même, à partir de *MD*, *ME* est *importée* avec comme préfixes de renommage successifs *f3* puis *e4*.

On construit maintenant l'exemple 2 à partir de l'exemple 1. Désormais *MD importe* également une nouvelle instance de *MC* renommée *c2.MC*. Le graphe ci-dessous indique quelles sont les instances effectivement *vues* par la nouvelle instance de *MA* créée transitivement par ce renommage.

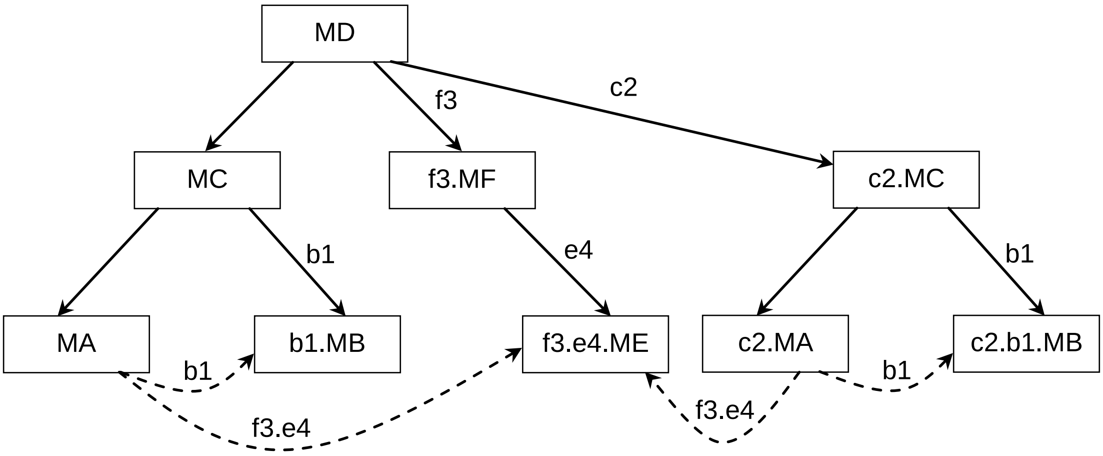


Figure 2 : exemple n°2 de SEES et renommage

La nouvelle instance de *MA* notée *c2.MA voit* les instances *c2.b1.MB* et *f3.e4.ME*. En effet, *c2.MC* est le premier ancêtre commun de *c2.MA* et de *c2.b1.MB*. À partir de *c2.MC*, *c2.b1.MB* est *importée* avec comme préfixe de renommage *b1*. De même, *MD* est l'ancêtre commun de *c2.MA* et de *ME*. À partir de *MD*, *ME* est *importée* avec comme préfixe de renommage *f3.e4*.

Visibilité

Soit M_A une machine abstraite ou un raffinement qui *voit* une instance de machine M_B. Les paramètres formels de M_B ne sont pas accessibles dans M_A. Les ensembles et les constantes de M_B sont accessibles dans les clauses INCLUDES, EXTENDS, PROPERTIES, INVARIANT, ASSERTIONS et dans le corps de l'initialisation et des opérations du composant. Les variables sont accessibles en lecture dans le corps de l'initialisation et des opérations. Il est possible d'utiliser les opérations de consultation (ne modifiant pas les variables) de M_B dans l'initialisation et dans les opérations de M_A.

Si l'instance de machine *vue* est renommée, alors ses variables et ses opérations sont accessibles dans la machine en préfixant leur nom par le préfixe de renommage de la machine *vue*.

Soit M_{A_i} une implantation qui *voit* une instance de machine M_B. Les paramètres formels de M_B ne sont pas accessibles dans M_{A_i}. Les ensembles et les constantes concrètes de M_B sont accessibles dans les clauses IMPORTS, EXTENDS, PROPERTIES, VALUES, INVARIANT, ASSERTIONS et dans le corps de l'initialisation et des opérations du composant. Les constantes abstraites de M_B sont en outre accessibles dans les clauses PROPERTIES, INVARIANT, ASSERTIONS et dans les variants et invariants de boucles, et les prédicats de substitutions ASSERT des opérations et de l'initialisation. Les variables concrètes de M_B sont accessibles en lecture dans le corps de l'initialisation et des opérations. Les variables abstraites de M_B sont en outre accessibles dans les variants et invariants de boucles, et les prédicats de substitutions ASSERT des opérations et de l'initialisation. Il est possible d'utiliser les opérations de consultation (ne modifiant pas les variables) de M_B dans l'initialisation et dans les opérations de M_{A_i}.

Transitivité

La clause SEES n'est pas transitive. Si un composant M_1 *voit* une machine M_2 qui elle-même *voit* une machine M_3, alors les constituants de M_3 ne sont pas accessibles par M_1. Si on veut qu'ils le soient, il faut que M_1 *voie* aussi explicitement M_3.

Tables de visibilité

Soit M_A une machine ou un raffinement qui *voit* une machine M_B. La table de visibilité suivante précise pour chaque constituant de M_B, les modes d'utilisation applicables dans les clauses de M_A.

Clauses de M_A / Constituants de M_B	CONSTRAINTS	Paramètres d'INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS
Paramètres formels					
Ensembles, énumérés littéraux, constantes concrètes		visible	visible	visible	visible
Constantes abstraites		visible	visible	visible	visible
Variables concrètes					visible – non modifiable
Variables abstraites					visible – non modifiable
Opérations					visible – non modifiable

Soit M_N une implantation qui *voit* une instance de machine M_B. La table de visibilité suivante précise pour chaque constituant de M_B, les modes d'utilisation applicables dans les clauses de M_N.

Clauses de M_N / Constituants de M_B	Paramètres d'IMPORTS / EXTENDS	PROPERTIES	VALUES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS		LOCAL_ OPERATIONS
					Instructions	Variants et invariants de boucles, prédicats d'ASSERT	
Paramètres formels							
Ensembles, énumérés littéraux, constantes concrètes	visible	visible	visible	visible	visible	visible	visible
Constantes abstraites		visible		visible		visible	visible
Variables concrètes					visible – non modifiable	visible	visible – non modifiable
Variables abstraites						visible	visible – non modifiable
Opérations					visible – non modifiable		visible – non modifiable

7.9 La clause INCLUDES

Syntaxe

Clause_includes ::=
"INCLUDES" (*Ident_ren* ["(" *Instanciation*$^{+","}$ ")"])$^{+","}$

Instanciation :=
Terme
| *Ensemble_nombres*
| "BOOL"
| *Intervalle*

Règle de typage

Dans le cas où une instance de machine *incluse* possède des paramètres, les paramètres effectifs doivent respecter la règle suivante. Les paramètres effectifs correspondant à des données scalaires doivent être de même type que les paramètres formels de la machine *incluse* et les paramètres effectifs correspondant à des ensembles doivent être de type $\mathbb{P}(T)$ où T est un type de base autre que STRING.

Restrictions

1. Les identificateurs d'une clause INCLUDES (sans tenir compte des éventuels préfixes de renommage) doivent désigner des machines abstraites.
2. Les identificateurs renommés ne peuvent avoir au plus qu'un préfixe de renommage.
3. Chaque nom de machine doit être suivi d'une liste de paramètres effectifs de même nombre que les paramètres de la machine *incluse*.

Description

La clause INCLUDES permet de regrouper dans une machine abstraite ou un raffinement, les constituants (ensembles, constantes et variables) d'instances de machines ainsi que leurs propriétés (clause PROPERTIES et INVARIANT), afin de créer un composant enrichi à l'aide d'autres machines abstraites. Ce lien permet de construire de manière modulaire des machines abstraites ou des raffinements.

Utilisation

La clause INCLUDES contient la déclaration de la liste des instances de machines *incluses*. Il s'agit soit du nom de la machine seul, qui référence l'instance sans renommage de la machine (l'instance de la machine est alors confondue avec celle-ci), soit du nom de la machine précédé d'un renommage, qui désigne l'instance de la machine renommée (dans ce cas on peut avoir plusieurs instances de la machine, chaque instance correspondant à un préfixe distinct). Si une instance de machine *incluse* est paramétrée, les paramètres effectifs de la machine doivent être fournis, ce qui permet d'instancier les paramètres formels de la nouvelle instance de machine. Les paramètres d'une machine abstraite sont décrits dans la clause CONSTRAINTS (cf. §7.5 *La clause CONSTRAINTS*). Ce sont soit des scalaires, soit des ensembles de scalaires. Une Obligation de Preuve est générée afin de prouver que les paramètres effectifs d'une instance de machine *incluse* respectent les contraintes de la machine.

Instanciation des paramètres scalaires

Un paramètre effectif scalaire d'une instance de machine *incluse* est de type $\mathbb{Z}$, BOOL, ou *Ens*, si *Ens* est un ensemble abstrait ou énuméré.

Instanciation des paramètres ensembles

Un paramètre effectif ensemble d'une instance de machine *incluse* est de type $\mathbb{P}(\mathbb{Z})$, $\mathbb{P}$(BOOL), ou $\mathbb{P}(Ens)$, si *Ens* est un ensemble abstrait ou énuméré.

Raffinement d'un composant qui *inclut*

Lorsque l'abstraction d'un raffinement M_{A_r} *inclut* une instance de machine M_B, alors, le raffinement M_{A_r} peut à nouveau *inclure* M_B. Les ensembles abstraits ou énumérés, les constantes concrètes et les variables concrètes de M_{A_r} qui proviennent de l'*inclusion* précédente de M_B sont collés par homonymie avec ceux de M_B. Dans le raffinement M_{A_r}, toutes ces données sont alors considérées comme provenant de M_B, et non pas comme héritées de l'abstraction de M_{A_r}. Les constantes abstraites et les variables abstraites de l'abstraction de M_{A_r} qui proviennent de l'*inclusion* précédente de M_B sont raffinées par les données homonymes de M_B.

Implantation d'un composant qui *inclut*

Lorsqu'une machine abstraite ou un raffinement *MA* *inclut* une instance de machine *MB*, alors plusieurs possibilités peuvent être envisagées lors de l'écriture de l'implantation *MA_i* de *MA*. L'implantation peut *importer* l'instance de machine *incluse* par *MA*. Dans ce cas, les constituants de *MB* regroupés dans *MA* lors de l'*inclusion* sont implantés dans *MA_i* par homonymie avec ceux de l'instance de machine *importée*. Sinon, l'implantation peut ne pas *importer* l'instance de machine *MB*. Elle doit alors implanter les constituants de *MB* regroupés dans *MA* lors de l'*inclusion* soit directement, soit avec des constituants d'instances de machines *vues* ou *importées*. L'utilisateur reste libre de choisir le découpage qui lui convient. Dans le deuxième cas, si aucune instance de *MB* n'est *importée* dans le projet alors *MB* s'appelle un module abstrait (cf. §8.2 *Module B*). L'instance locale de *MB* est alors créée pour servir d'intermédiaire de spécification et elle est abandonnée dans la suite du développement.

Visibilité

Les paramètres formels des machines *incluses* ne sont pas accessibles dans le composant qui *inclut*. Les ensembles et les constantes sont accessibles dans les clauses PROPERTIES, INVARIANT, ASSERTIONS et dans le corps de l'initialisation et des opérations de la machine. Les variables sont accessibles dans les invariants et les assertions. Elles sont, en outre, accessibles en lecture dans le corps de l'initialisation et des opérations. Il est possible d'utiliser les opérations d'une machine *incluse* dans l'initialisation et dans les opérations de la machine.

Transitivité

La clause INCLUDES est transitive : si un composant M_1 *inclut* une instance de machine M_2 qui elle-même *inclut* une instance de machine M_3, alors les ensembles, les constantes, les variables et leurs propriétés de M_3 sont regroupés avec ceux de M_2 qui sont eux-mêmes regroupés avec ceux de M_1. Ces constituants sont donc accessibles par M_1. Par contre, les opérations de M_3 ne sont pas accessibles par M_1. Les propriétés de regroupement et d'accès s'étendent pour un nombre quelconque de machines transitivement *incluses*.

Regroupement des données

Si un composant M_A *inclut* des instances de machines M_{inc}, alors, vis-à-vis de l'extérieur du composant, l'ensemble des constituants (les ensembles, les constantes et les variables) des machines *incluses* et transitivement *incluses* fait partie du composant M_A au même titre que les constituants propres de ce composant. Ainsi, si un composant M_B *voit* M_A, les ensembles, les constantes et les variables des machines *incluses* et transitivement *incluses* par M_A sont accessibles dans le composant M_B selon les mêmes règles que les ensembles, les constantes et les variables de M_A. Si un composant M_{A_r} raffine M_A, les ensembles, les constantes et les variables des machines *incluses* et transitivement *incluses* par M_A sont accessibles dans le composant M_{A_r} selon les mêmes règles que les ensembles, les constantes et les variables propres de M_A.

Promotion des opérations

Les opérations d'une machine *incluse* peuvent devenir automatiquement des opérations du composant qui réalise l'*inclusion*. Il s'agit du mécanisme de promotion d'opération (cf. §7.10 *La clause PROMOTES* et §7.11 *La clause EXTENDS*).

Table de visibilité

Soit M_A une machine ou un raffinement qui *inclut* une instance de machine M_B. La table de visibilité suivante précise pour chaque constituant de M_B, les modes d'utilisation applicables dans les clauses de M_A.

Clauses de M_A / Constituants de M_B	CONSTRAINTS	Paramètres d'INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS
Paramètres formels					
Ensembles, énumérés littéraux, constantes concrètes			visible	visible	visible
Constantes abstraites			visible	visible	visible
Variables concrètes				visible	visible – non modifiable
Variables abstraites				visible	visible – non modifiable
Opérations					visible modifiable

On note que ces liens de visibilité sont les mêmes que ceux que l'on aurait eu si les clauses de la machine *incluse* avaient été regroupées dans les clauses correspondantes de la machine *incluante* (cf. schéma de regroupement ci-dessous).

Machine équivalente

Si *MA* est une machine abstraite qui *inclut MB*, les deux machines sont logiquement équivalentes à une machine unique ayant les caractéristiques suivantes :

- le nom de la machine équivalente, ses paramètres formels et ses opérations sont ceux de *MA*,
- ses ensembles abstraits et énumérés, ses constantes, ses variables, ses propriétés, son invariant, ses assertions et son initialisation sont respectivement ceux de *MB* concaténés avec ceux de *MA*. Dans le cas de l'initialisation, la substitution équivalente est le séquencement de l'initialisation de *MB* puis de *MA*. En effet,

comme les opérations de *MB* peuvent être appelées dans l'initialisation de *MA*, il faut que l'invariant de *MB* ait déjà été établi,

- dans l'initialisation et dans le corps des opérations de la machine équivalente, les appels des opérations de *MB* sont remplacées par leur corps, en appliquant la définition de la substitution appel d'opération (cf. §6.16 *Substitution appel d'opération*),
- lors de l'*inclusion* dans la machine équivalente de l'initialisation, de la clause ASSERTIONS et éventuellement des opérations de *MB*, les paramètres formels de *MB* sont remplacés par leurs instanciations.

```
MACHINE
    MA ( ParamA )
CONSTRAINTS
    ConstrA
INCLUDES
    MB ( Inst )
SETS
    SetsA
CONCRETE_CONSTANTS
    ConcCteA
ABSTRACT_CONSTANTS
    AbsCteA
PROPERTIES
    PropA
CONCRETE_VARIABLES
    ConcVarA
ABSTRACT_VARIABLES
    AbsVarA
INVARIANT
    InvA
ASSERTIONS
    AssertA
INITIALISATION
    InitA
OPERATIONS
    opA1 = SubstA1 ;
    …
    opAn = SubstAn
END
```

```
MACHINE
    MB ( ParamB )
CONSTRAINTS
    ConstrB


SETS
    SetsB
CONCRETE_CONSTANTS
    ConcCteB
ABSTRACT_CONSTANTS
    AbsCteB
PROPERTIES
    PropB
CONCRETE_VARIABLES
    ConcVarB
ABSTRACT_VARIABLES
    AbsVarB
INVARIANT
    InvB
ASSERTIONS
    AssertB
INITIALISATION
    InitB
OPERATIONS
    opB1 = SubstB1 ;
    ...
    opBm = SubstBm
END
```

⇔

```
MACHINE
    MA ( ParamA )
CONSTRAINTS
    ConstrA


SETS
    SetsB ; SetsA
CONCRETE_CONSTANTS
    ConcCteB , ConcCteA
ABSTRACT_CONSTANTS
    AbsCteB , AbsCteA
PROPERTIES
    PropB ∧ PropA
CONCRETE_VARIABLES
    ConcVarB , ConcVarA
ABSTRACT_VARIABLES
    AbsVarB , AbsVarA
INVARIANT
    InvB ∧ InvA
ASSERTIONS
    AssertB ; AssertA
INITIALISATION
    InitB ; InitA
OPERATIONS
    opA1 = SubstA1 ;
    ...
    opAn = SubstAn
END
```

7.10 La clause PROMOTES

Syntaxe

Clause_promotes ::= "PROMOTES" $Ident\_ren^{+","}$

Restrictions

1. Les noms d'opérations promues par un composant doivent désigner des opérations d'instances de machines *incluses*, si le composant est une machine abstraite ou un raffinement, et d'instances de machines *importées*, si le composant est une implantation.
2. Chaque opération promue d'un raffinement d'une machine abstraite doit porter le même nom qu'une opération de la machine abstraite. Les deux opérations doivent alors avoir la même signature (leurs paramètres formels doivent porter le même nom, être dans le même ordre et avoir les mêmes types).

Description

La clause PROMOTES permet à un composant de promouvoir des opérations (cf. §7.23 *La clause OPERATIONS*) appartenant à des instances de machines créées par le composant. Il peut s'agir soit d'instances de machines *incluses*, si le composant est une machine abstraite ou un raffinement, soit d'instances de machines *importées* si le composant est une implantation. Promouvoir une opération d'une instance de machine M_B dans un composant M_A équivaut à définir dans M_A une opération dont le nom est celui de l'opération de M_B (éventuellement précédé du préfixe de renommage de M_B, si M_B est renommée), et dont la signature et le corps sont ceux de l'opération de M_B.

Utilisation

Chaque nom de la liste PROMOTES désigne le nom d'une opération d'une instance de machine *incluse* ou *importée*. Si l'instance de machine *incluse* ou *importée* est renommée, alors le nom de l'opération doit être précédé du préfixe de renommage de l'instance. Le nom, la signature et le service offert par une opération promue sont identiques au nom, à la signature et au service de l'opération qu'elle promeut.

Les opérations promues deviennent des opérations à part entière de la machine abstraite. Du point de vue des composants qui utilisent cette machine, rien ne distingue les opérations promues des opérations propres définies dans la clause OPERATIONS.

Contrairement aux opérations propres, les opérations promues d'une machine peuvent être appelées dans les clauses INITIALISATION et OPERATIONS de la machine, puisqu'il s'agit en réalité d'opérations de machines *incluses* ou *importées*.

Exemple

```
MACHINE
    MA
INCLUDES
    MB,
    r2.MC
PROMOTES
    opB1,
    opB3,
    r2.opC1,
    r2.opC3
...
END
```

```
MACHINE
    MB
OPERATIONS
    opB1 = ... ;
    opB3 = ...
...
END
```

```
MACHINE
    MC
OPERATIONS
    opC1 = ... ;
    opC3 = ...
...
END
```

Dans l'exemple ci-dessus, la machine *MA* *inclut* les instances de machines *MB* et *r2.MC*, puis elle promeut les opérations `opB1` et `opB3` de l'instance *MB* et les opérations `opC1` et `opC2` de l'instance renommée *r2.MC*. La machine possède désormais, en plus de ses propres opérations, les quatre opérations : `opB1`, `opB3`, `r2.opC1` et `r2.opC3`.

7.11 La clause EXTENDS

Syntaxe

Clause_EXTENDS ::= "EXTENDS" (*Ident_ren* ["(" *Instanciation*$^{+","}$ ")"])$^{+","}$

Clause_EXTENDS_B0 ::= "EXTENDS" (*Ident_ren* ["(" *Instanciation_B0*$^{+","}$ ")"])$^{+","}$

Description

Dans une machine abstraite ou un raffinement, la clause EXTENDS est équivalente à l'*inclusion* (cf. §7.9 *La clause INCLUDES*) d'instances de machines et à la promotion (cf. §7.10 *La clause PROMOTES*) de toutes les opérations des instances de machines *incluses*.

Dans une implantation, la clause EXTENDS est équivalente à l'*importation* (cf. §7.7 *La clause IMPORTS*) d'instances de machines et à la promotion (cf. §7.10 *La clause PROMOTES*) de toutes les opérations des instances de machines *importées*.

Restrictions

(cf. §7.9 *La clause INCLUDES* et §7.7 *La clause IMPORTS*)

7.12 La clause USES

Syntaxe

Clause_uses ::= "USES" *Ident_ren*$^{+","}$

Restrictions

1. Si une machine *MA* utilise une instance de machine *Mused*, alors il doit exister dans le projet une machine qui *inclut* une instance de *MA* et l'instance *Mused* (cf. §8.3 Règle n°9 sur les liens USES).
2. Une machine qui *utilise* d'autres machines ne doit pas être raffinée. Elle doit donc constituer un module abstrait (cf. §8.2 Module B). Elle ne doit pas être *vue* ni *importée* par d'autres composants.

Description

Lorsqu'un composant *inclut* plusieurs machines, les machines *incluses* peuvent partager les données d'une des machines *incluses* en *utilisant* (par un lien USES) cette machine *incluse*. Dans le schéma ci-dessous, la machine *MA inclut* les instances de machines *MB*, *MC* et *MD*. La machine *incluse MC* est *utilisée* (lien USES) par *MB* et *MD*, ce qui permet à *MB* et à *MD* d'accéder aux données de *MC*. Les données de *MC* sont donc partagées par *MB* et *MD*.

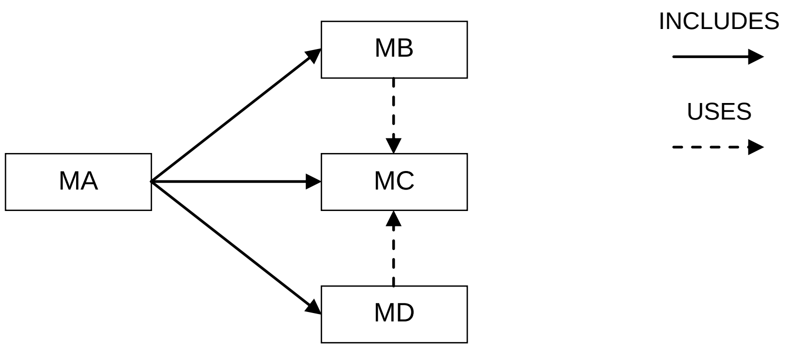


Figure 3 : principes du lien USES

Utilisation

Soit *MA* une machine abstraite, qui *utilise* d'autres machines. Les noms de la liste USES désignent des instances de machines *utilisées* par *MA*. Une instance de *MA*, ainsi que les instances de machines *utilisées* doivent toutes être *incluses* par un composant unique.

Visibilité

Les paramètres formels des machines *utilisées* sont accessibles dans les clauses PROPERTIES, INVARIANT, ASSERTIONS et dans le corps de l'initialisation et des opérations de la machine qui *utilise*. Les ensembles et les constantes sont accessibles dans les clauses PROPERTIES, INVARIANT, ASSERTIONS et dans le corps de l'initialisation et des opérations de la machine. Les variables sont accessibles dans les invariants et les assertions. Elles sont, en outre, accessibles en lecture dans le corps de l'initialisation et des opérations. Il est interdit d'appeler les opérations d'une machine *utilisée* dans l'initialisation et dans les opérations de la machine.

Transitivité

La clause USES n'est pas transitive. Si une machine M_1 *utilise* une machine M_2 qui elle-même *utilise* une machine M_3, alors les paramètres formels, les ensembles, les constantes et les variables de M_3 ne sont pas accessibles par M_1.

Exemple

```
MACHINE
    MA
INCLUDES
    MX,
    y2.MY,
    z3.MZ
...
END
```

```
MACHINE
    MX
USES
    y2.MY
...
END
```

```
MACHINE
    MY

...
END
```

```
MACHINE
    MZ
INCLUDES
    y2.MY
...
END
```

La machine *MA* *inclut* les instances de machines *MX*, *y2.MY* et *z3.MZ*. L'instance de machine *y2.MY* est *utilisée* par les machines *MX* et *MZ* (et donc par les instances de machines *MX* et *z3.MZ*).

Table de visibilité

Soit M_A une machine qui *utilise* une machine M_B. La table de visibilité suivante précise pour chaque constituant de M_B, les modes d'utilisation applicables dans les clauses de M_A.

Clauses de M_A / Constituants de M_B	CONSTRAINTS	Paramètres d'INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS
Paramètres formels				visible	visible
Ensembles, énumérés littéraux, constantes concrètes			visible	visible	visible
Constantes abstraites			visible	visible	visible
Variables concrètes				visible	visible – non modifiable
Variables abstraites				visible	visible – non modifiable
Opérations					

7.13 La clause SETS

Syntaxe

Clause_sets	::=	"SETS" *Ensemble*$^{+";"}$
Ensemble	::=	Ident
	|	Ident "=" "{" Ident$^{+","}$ "}"

Restrictions

1. Le nom d'un ensemble abstrait ou énuméré d'un raffinement doit différer respectivement du nom des ensembles abstraits ou énumérés des machines abstraites *vues* ou *incluses* par le raffinement, sauf dans le cas suivant : un ensemble abstrait ou énuméré appartenant à une abstraction du raffinement peut être identique respectivement à un ensemble abstrait ou énuméré d'une machine *vue* ou *incluse* par le raffinement. Dans le cas de l'ensemble énuméré, les deux ensembles énumérés doivent alors avoir le même nom et la même liste d'éléments énumérés dans le même ordre.
2. Le nom d'un ensemble abstrait ou énuméré d'une implantation doit différer respectivement du nom des ensembles abstraits ou énumérés des machines abstraites *vues* ou *importées* par l'implantation, sauf dans le cas suivant : un ensemble abstrait ou énuméré appartenant à une abstraction de l'implantation peut être identique respectivement à un ensemble abstrait ou énuméré d'une machine *vue* ou *importée* par l'implantation. Dans le cas de l'ensemble énuméré, les deux ensembles énumérés doivent alors avoir le même nom et la même liste ordonnée d'éléments énumérés.

Description

La clause SETS définit la liste des ensembles abstraits et des ensembles énumérés d'un composant.

Utilisation

Les ensembles abstraits et les ensembles énumérés définissent des types de base (cf. §3.1 *Fondements du typage*). Les ensembles abstraits et les ensembles énumérés définis dans un composant sont des données concrètes. Ils sont implicitement conservés au cours du raffinement du composant, jusqu'à l'implantation.

- Un ensemble abstrait est défini par son nom. Les ensembles abstraits sont utilisés pour désigner des objets dont on ne veut pas définir la structure au niveau d'une abstraction. Tout ensemble abstrait est implicitement fini et non vide. Il devra être valué dans l'implantation du composant (cf. §7.17 *La clause VALUES*). À terme, tout ensemble abstrait est valué par un intervalle entier fini non vide, mais pas par un ensemble énuméré.
- Un ensemble énuméré est défini par son nom et par la liste ordonnée et non vide de ses éléments énumérés. Les ensembles énumérés servent à décrire une énumération. Les éléments d'un ensemble énuméré sont appelés des énumérés littéraux. Ils possèdent la même sémantique que des constantes concrètes dont le type est l'ensemble énuméré.

Liste des ensembles abstraits et énumérés d'un composant

Les ensembles abstraits et les ensembles énumérés d'une machine abstraite regroupent les ensembles définis dans la machine ou provenant des machines *incluses* par la machine.

Les ensembles abstraits et les ensembles énumérés d'un raffinement regroupent les ensembles définis dans le raffinement, provenant de l'abstraction du raffinement ou provenant des machines *incluses* par le raffinement.

Les ensembles abstraits et les ensembles énumérés d'une implantation regroupent les ensembles définis dans l'implantation et ceux provenant de l'abstraction de l'implantation.

Visibilité

Les ensembles abstraits, les ensembles énumérés et les éléments d'ensembles énumérés d'une machine abstraite sont accessibles dans la machine, au sein des clauses INCLUDES, EXTENDS, PROPERTIES, INVARIANT, ASSERTIONS, INITIALISATION et OPERATIONS. Ils sont également accessibles par les composants qui *voient*, *incluent*, *utilisent* ou *importent* la machine.

Les ensembles abstraits, les ensembles énumérés et les éléments d'ensembles énumérés d'un raffinement sont accessibles dans le raffinement, au sein des clauses INCLUDES, EXTENDS, PROPERTIES, INVARIANT, ASSERTIONS, INITIALISATION et OPERATIONS.

Les ensembles abstraits, les ensembles énumérés et les éléments d'ensembles énumérés d'une implantation sont accessibles dans l'implantation, au sein des clauses IMPORTS, EXTENDS, PROPERTIES, VALUES, INVARIANT, ASSERTIONS, INITIALISATION et OPERATIONS.

Exemple

```
MACHINE
    MA
SETS
    POSITION ;
    MARCHE = {Arret, Avant, Arriere} ;
    DIRECTION = {Nord, Sud, Est, Ouest}
...
END
```

```
IMPLEMENTATION
    MA_i
REFINES
    MA
IMPORTS
    MB
SETS
    VITESSE ;
    SIGNAL = {Rouge, Orange, Vert}
VALUES
    POSITION = 0 .. 100 ;
    VITESSE = -10 .. 10
...
END
```

```
MACHINE
    MB
SETS
    MARCHE = {Arret, Avant, Arriere}
...
END
```

Dans l'exemple ci-dessus, la machine abstraite *MA* définit l'ensemble abstrait *POSITION* et les ensembles énumérés *MARCHE* et *DIRECTION*. *MA_i*, l'implantation de *MA* définit un nouvel ensemble abstrait *VITESSE* et un nouvel ensemble énuméré *SIGNAL*. Les deux ensembles abstraits de *MA_i* sont valués dans la clause *VALUES*. L'implantation *MA_i* *importe* la machine *MB* qui possède un ensemble énuméré *MARCHE* identique à celui de *MA_i*, puisqu'il a le même nom et la même liste ordonnée d'éléments énumérés.

7.14 La clause CONCRETE_CONSTANTS

Syntaxe

Clause_concrete_constants ::=
"CONCRETE_CONSTANTS" Ident$^{+","}$
| "CONSTANTS" Ident$^{+","}$

Description

La clause CONCRETE_CONSTANTS définit la liste des constantes concrètes d'un composant.

Une constante concrète est une donnée implémentable dans un langage informatique dont la valeur reste constante et qui est implicitement conservée au cours du raffinement jusqu'à l'implantation. Une constante concrète peut être un entier concret, un booléen, un élément d'un ensemble abstrait ou d'un ensemble énuméré, un intervalle fini et non vide d'entiers concrets, un intervalle fini et non vide d'ensemble abstrait, ou un tableau concret (cf. §3.4 *Types et contraintes des données concrètes*).

Restriction

1. Le nom d'une nouvelle constante concrète d'un raffinement ou d'une implantation doit différer du nom des constantes (concrètes ou abstraites) de l'abstraction, sauf dans le cas suivant : une constante concrète peut raffiner une constante abstraite homonyme de l'abstraction. Les deux constantes sont alors implicitement égales et elles doivent être de même type.

Utilisation

Les noms de clause CONCRETE_CONSTANTS et CONSTANTS sont équivalents, ils peuvent donc être utilisés indifféremment.

Les constantes concrètes d'une machine abstraite regroupent les constantes concrètes définies dans la machine ou provenant de machines *incluses* (cf. §7.9 *La clause INCLUDES*). Les constantes concrètes d'un raffinement regroupent les constantes concrètes définies dans le raffinement, provenant de l'abstraction et provenant d'*inclusions*. Les constantes concrètes d'une implantation regroupent les constantes concrètes définies dans l'implantation et provenant de l'abstraction.

Le typage et les propriétés des constantes concrètes sont exprimés dans la clause PROPERTIES (cf. §7.16 *La clause PROPERTIES*).

Chaque constante concrète appartenant à un composant doit être valuée dans l'implantation du composant (cf. §7.17 *La clause VALUES*).

Chaque constante concrète définie dans une machine abstraite doit être typée dans la clause PROPERTIES de la machine. Une constante concrète définie dans un raffinement ou dans une implantation peut être :

- soit une nouvelle constante concrète. Elle doit alors être typée explicitement dans la clause PROPERTIES du raffinement ou de l'implantation.
- soit le raffinement d'une constante abstraite, si son nom est identique à celui d'une constante abstraite de l'abstraction. Pour cela, la constante abstraite doit être de même nature qu'une constante concrète. Elle est alors typée implicitement par

un prédicat de liaison qui signifie que la constante concrète est égale à la constante abstraite homonyme.

Visibilité

Les constantes concrètes d'une machine abstraite sont accessibles en lecture dans les clauses PROPERTIES, INVARIANT, ASSERTIONS, INCLUDES, EXTENDS et dans le corps de l'initialisation et des opérations de la machine. Elles sont accessibles en lecture par les composants qui *voient*, *incluent*, *utilisent* ou *importent* cette machine.

Les constantes concrètes d'un raffinement sont accessibles en lecture dans les clauses PROPERTIES, INVARIANT, ASSERTIONS, INCLUDES, EXTENDS et dans le corps de l'initialisation et des opérations du raffinement.

Les constantes concrètes d'une implantation sont accessibles en lecture dans les clauses PROPERTIES, INVARIANT, ASSERTIONS, IMPORTS, EXTENDS et dans le corps de l'initialisation et des opérations de l'implantation. Elles sont en outre accessibles en écriture dans la clause VALUES.

Exemple

```
MACHINE
    MA
CONCRETE_CONSTANTS
    PosMin,
    PosMax
ABSTRACT_CONSTANTS
    PosInit
PROPERTIES
    PosMin ∈ INT ∧
    PosMax ∈ NAT ∧
    PosInit ∈ INT
...
END
```

```
IMPLEMENTATION
    MA_i
REFINES
    MA
CONCRETE_CONSTANTS
    PosMoyenne,
    PosInit
PROPERTIES
    PosMoyenne ∈ INT
VALUES
    PosMin = -100 ;
    PosMax = 100 ;
    PosMoyenne = 0 ;
    PosInit = 50
...
END
```

La machine abstraite *MA* définit deux constantes concrètes *PosMin* et *PosMax* et une constante abstraite *PosInit*. L'implantation *MA_i* de *MA* définit les nouvelles constantes concrètes *PosMoyenne* et *PosInit*. Cette dernière raffine la constante abstraite homonyme de *MA*. Toutes les constantes de *MA_i* (*PosMin*, *PosMax*, *PosMoyenne* et *PosInit*) sont valuées dans l'implantation.

7.15 La clause ABSTRACT_CONSTANTS

Syntaxe

Clause_abstract_constants ::= "ABSTRACT_CONSTANTS" Ident$^{+","}$

Description

La clause ABSTRACT_CONSTANTS contient la liste des constantes abstraites d'une machine abstraite ou d'un raffinement.

Une constante abstraite est une donnée de valeur constante qui sera raffinée dans le raffinement du composant.

Restriction

1. Le nom d'une constante abstraite d'un raffinement doit différer du nom des constantes (concrètes ou abstraites) de l'abstraction, sauf dans le cas suivant : une constante abstraite peut raffiner une constante abstraite homonyme de l'abstraction, elle est alors implicitement égale à la constante abstraite de l'abstraction.

Utilisation

Le nom de la clause est suivi par une liste d'identificateurs qui représentent le nom des constantes abstraites. Le typage et la déclaration des propriétés des constantes abstraites sont effectués dans la clause PROPERTIES (cf. §7.16 *La clause PROPERTIES*).

Chaque constante abstraite définie dans un raffinement peut être :

- soit une nouvelle constante abstraite. Elle doit être typée et peut éventuellement être contrainte dans la clause PROPERTIES.
- soit le raffinement d'une constante abstraite. Si son nom est identique à celui d'une constante abstraite de l'abstraction, il est alors inutile de typer la constante abstraite dans la clause PROPERTIES, car elle est typée par défaut à l'aide d'une propriété de liaison implicite qui signifie que la nouvelle constante abstraite est égale à la constante abstraite de l'abstraction. D'autres propriétés sur la constante peuvent être exprimées dans la clause PROPERTIES.

Si M_n raffine M_{n-1}, les constantes abstraites de M_{n-1} peuvent également être raffinées en tant que constantes concrètes de M_n (cf. §7.14 *La clause CONCRETE_CONSTANTS*). Si une constante abstraite de M_{n-1} n'est pas raffinée en tant que constante abstraite, ni en tant que constante concrète, alors elle ne fait plus partie des constantes de M_n. On dit alors qu'elle disparaît dans M_n.

La clause ABSTRACT_CONSTANTS est interdite en implantation. En effet, contrairement aux constantes concrètes, les constantes abstraites ne sont pas systématiquement implémentables dans un langage informatique.

Visibilité

Les constantes abstraites d'une machine sont accessibles en lecture dans les clauses INCLUDES, EXTENDS, PROPERTIES, INVARIANT, ASSERTIONS, INITIALISATION et OPERATIONS de la machine. Elles sont accessibles en lecture par les composants qui *importent*, *voient*, *incluent* ou *utilisent* cette machine (cf. Annexe C. *Tables de visibilité*).

Les constantes abstraites d'un raffinement sont accessibles en lecture dans les clauses INCLUDES, EXTENDS, PROPERTIES, INVARIANT, ASSERTIONS, INITIALISATION et OPERATIONS du raffinement.

Exemple

```
MACHINE
    Reseau
ABSTRACT_CONSTANTS
    NbrAbonnesMax,
    Connexion
PROPERTIES
    NbrAbonnesMax ∈ NAT ∧
    Connexion ∈ 0 .. NbrAbonnesMax ↔ 0 .. NbrAbonnesMax
...
END
```

```
REFINEMENT
    Reseau_r
REFINES
    Reseau
ABSTRACT_CONSTANTS
    Connexion,
    AbonnesInit
PROPERTIES
    AbonnesInit ⊂ NAT
...
END
```

La machine abstraite *Reseau* définit deux constantes abstraites *NbrAbonnesMax* et *Connexion*. *Reseau_r*, le raffinement de *Reseau*, définit la nouvelle constante abstraite *AbonnesInit* et la constante *Connexion*, homonyme à une constante abstraite de la machine abstraite *Reseau*. Comme *Connexion* est implicitement égale à la constante de l'abstraction, sa redéfinition permet de conserver cette constante abstraite et ses propriétés dans le raffinement.

7.16 La clause PROPERTIES

Syntaxe

Clause_properties ::= "PROPERTIES" *Prédicat*

Description

La clause PROPERTIES permet de typer les constantes (concrètes et abstraites) définies dans un composant et d'exprimer des propriétés sur ces constantes.

Restrictions

1. Chaque constante, concrète ou abstraite, définie dans un composant et qui n'est pas homonyme à une constante de l'éventuelle abstraction du composant, doit être typée dans la clause PROPERTIES du composant par un prédicat de typage (cf. §3.4 *Typage des constantes concrètes* et §3.3 *Typage des données abstraites*) situé au plus haut niveau d'analyse syntaxique dans une série de conjonctions. Ces constantes ne peuvent pas être utilisées dans la clause PROPERTIES avant d'avoir été typées.
2. Chaque constante définie dans un raffinement ou une implantation et qui est homonyme à une constante de l'abstraction n'a pas à être typée, puisqu'elle est typée implicitement par un prédicat qui signifie que la nouvelle constante est égale à la constante homonyme de l'abstraction. Alors, les deux constantes homonymes doivent être de même type.
3. Si une constante d'une instance de machine *incluse* ou *vue* par un raffinement est homonyme et de même nature (abstraite ou concrète) à une constante de l'abstraction du raffinement, alors les deux constantes homonymes désignent la même donnée et elles doivent être de même type.
4. Si une constante d'une instance de machine *importée* ou *vue* par une implantation est homonyme et de même nature (abstraite ou concrète) à une constante de l'abstraction de l'implantation, alors les deux constantes homonymes désignent la même donnée et elles doivent être de même type.

Utilisation

La clause PROPERTIES d'une machine abstraite permet de typer les constantes de la machine et de définir leurs propriétés.

La clause PROPERTIES d'un raffinement permet de typer les nouvelles constantes du raffinement et de définir leurs propriétés et notamment celles qui explicitent les liens qui doivent exister nécessairement entre les constantes du raffinement et celles de l'abstraction. Chaque nouvelle constante doit être typée par un prédicat de typage. Il est inutile de typer une constante qui raffine une constante abstraite homonyme de l'abstraction, car elle est typée par défaut à l'aide d'un prédicat de liaison implicite qui signifie que la nouvelle constante est égale à la constante abstraite de l'abstraction.

L'utilisation de la clause PROPERTIES dans un raffinement et dans une implantation sont similaires. Cependant, comme la déclaration de constantes abstraites est interdite dans une implantation, seules les nouvelles constantes concrètes de l'implantation doivent être typées dans des prédicats de typage de la clause PROPERTIES.

Typage des constantes
Les constantes doivent être typées dans l'un des prédicats situés au plus haut niveau de l'analyse syntaxique de la clause PROPERTIES séparés par des conjonctions '$\wedge$', à l'aide de prédicats de typage de données abstraites pour les constantes abstraites (cf. §3.3 *Typage des données abstraites*) et de prédicats de typage de constantes concrètes pour les constantes concrètes (cf. §3.4 *Typage des constantes concrètes*).

Propriétés des constantes
Le prédicat de propriétés sur les constantes permet d'exprimer des propriétés générales portant sur les constantes.

Liaison entre constantes
La clause PROPERTIES d'un raffinement permet de préciser quels liens unissent les constantes déclarées dans le raffinement et les constantes de son abstraction. Chaque prédicat définissant un tel lien est appelé prédicat de liaison. Un prédicat de liaison peut aussi bien prendre la forme d'un prédicat de typage que d'une propriété générale. On rappelle également qu'un prédicat de liaison implicite lie par défaut deux constantes homonymes.

Une constante d'un raffinement ou d'une implantation peut être homonyme à une constante de même nature concrète ou abstraite d'une instance de machine *incluse* ou *importée*. Deux constantes homonymes fusionnent, elles représentent donc la même donnée et doivent alors être de même type.

Visibilité
Dans la clause PROPERTIES d'une machine, les ensembles et les constantes de la machine sont accessibles. Les ensembles et les constantes des machines *incluses* ou *vues* sont accessibles. Par contre les paramètres de la machine, de même que ceux des machines *utilisées* ne sont pas accessibles.

Exemple

```
MACHINE
    MA
CONCRETE_CONSTANTS
    Cte1
ABSTRACT_CONSTANTS
    Cte2
PROPERTIES
    Cte1 ∈ INT ∧
    Cte2 ∈ ℕ ∧
    (Cte1 < 0 ⇒ Cte2 = 0)
...
END
```

La machine abstraite *MA* définit la constante concrète *Cte1* et la constante abstraite *Cte2*. Ces deux constantes sont typées dans des prédicats de typage situés au plus haut niveau de l'analyse syntaxique dans une liste de conjonctions. L'implication qui suit ces prédicats de typage exprime une propriété portant sur *Cte1* et *Cte2*. Il faut noter que les parenthèses qui l'entourent sont nécessaires dans ce cas, à cause de la priorité de '$\Rightarrow$' plus faible que celle de '$\wedge$' (sans ces parenthèses, le prédicat serait analysé comme

$(Cte1 \in \mathsf{INT} \land Cte2 \in \mathbb{N} \land Cte1 < 0) \Rightarrow Cte2 = 0$; alors les prédicats de typage de *Cte1* et *Cte2* ne seraient plus dans une liste de conjonctions au plus haut niveau syntaxique).

7.17 La clause VALUES

Syntaxe

Clause_values ::= "VALUES" *Valuation*$^{+";"}$

Valuation ::= Ident "=" *Terme*
| Ident "=" *Expr_tableau*
| Ident "=" *Intervalle_B0*

Expr_tableau ::= Ident
| "{" (*Terme_simple*$^{+"\mapsto"}$ "$\mapsto$" *Terme*)$^{+","}$ "}"
| *Ensemble_simple*$^{+"\times"}$ "$\times$" "{" *Terme* "}"

Intervalle_B0 ::= *Expression_arithmétique* ".." *Expression_arithmétique*
| *Ensemble_nombres_B0*

Ensemble_nombres_B0 ::= "NAT"
| "NAT_1"
| "INT"

Ensemble_simple ::= *Ensemble_nombres_B0*
| "BOOL"
| *Intervalle_B0*
| Ident

Description

La clause VALUES permet de donner une valeur aux constantes concrètes (cf. §7.14 *La clause CONCRETE_CONSTANTS*) et aux ensembles abstraits (cf. §7.13 *La clause SETS*) de l'implantation.

Restrictions

1. Chaque constante concrète ou chaque ensemble abstrait qui n'est pas homonyme à une donnée de même nature d'une instance de machine *vue* ou *importée* doit être valué dans la clause VALUES une et une seule fois. Dans ce cas, on dit que la donnée est valuée explicitement, sinon on dit qu'elle est valuée implicitement.
2. Si une constante concrète est homonyme à une constante concrète d'une instance de machine *vue* ou *importée*, alors les deux constantes doivent avoir le même type.
3. Si une constante concrète ou un ensemble abstrait de l'implantation est utilisé en partie droite d'une valuation, alors il doit soit avoir été valué auparavant, soit être implanté par homonymie.

Utilisation

Le nom de la clause VALUES est suivi d'une liste de valuations. L'ordre des valuations est significatif à cause de la valuation des ensembles abstraits (cf. ci-dessous). Chaque valuation permet de donner explicitement une valeur à une constante concrète ou à un ensemble abstrait. Elle se compose du nom de la donnée à valuer, suivi de l'opérateur égal '=' et de la valeur de la donnée. Les valuations explicites se scindent en plusieurs parties : la valuation des constantes concrètes scalaires, des constantes tableaux, des constantes intervalles et des ensembles abstraits.

Valuation des ensembles abstraits

Lors de la valuation d'un ensemble abstrait *AbsSet*, le type que représente cet ensemble change. Il prend le type de l'ensemble qui le value.

Voici les différentes manières de valuer *AbsSet* :

- par un intervalle d'entiers non vide contenu dans INT. Le type *AbsSet* devient alors $\mathbb{Z}$,
- par un ensemble abstrait *AbsSet2* d'une machine *vue* ou *importée*. Le type *AbsSet* devient alors *AbsSet2*.
- implicitement, par un ensemble abstrait homonyme d'une machine *vue* ou *importée*. Le type *AbsSet* reste le même. En effet, comme les deux ensembles abstraits portent le même nom, ils sont de même type,
- par un ensemble abstrait *AbsSet3* de l'implantation qui a déjà été valué dans la clause VALUES ou qui est valué par homonymie avec un ensemble abstrait d'une machine *vue* ou *importée*. Le type *AbsSet* devient alors le type de *AbsSet3*.

Il faut prouver que les ensembles avec lesquels sont valués les ensembles abstraits sont finis et non vides. Si un ensemble abstrait est valué par un intervalle, il faut prouver que les bornes de l'intervalle appartiennent à INT et si les bornes de l'intervalle sont des expressions arithmétiques, il faut prouver que chaque sous-expression est bien définie et qu'elle appartient à INT (cf. §7.25.2 *Les termes*).

Les données de l'implantation dont le type est construit à l'aide du type *AbsSet* subissent également un changement de type. Cette modification consiste à remplacer dans le type des données chaque occurrence de *AbsSet* par le type qui sert à le valuer. La portée de ce changement de type comprend la partie de la clause VALUES, après valuation de *AbsSet* ainsi que les clauses IMPORTS, EXTENDS, PROPERTIES, INVARIANT, ASSERTION, INITIALISATION et OPERATIONS de l'implantation.

L'intérêt du changement de type réside principalement dans le cas de la valuation par un intervalle d'entiers (tout ensemble abstrait sera valué à terme par un intervalle d'entiers, les autres cas de valuation ne font que différer cette valuation). Lorsque *AbsSet* est valué par un intervalle d'entiers, les données appartenant à *AbsSet* peuvent recevoir des valeurs entières (en particulier des entiers littéraux) et être manipulées à l'aide des opérateurs arithmétiques, puisque ces opérateurs ne sont définis en B que pour des entiers. Elles deviennent donc concrètement utilisables.

Exemple de changement de type

```
MACHINE
    MA
SETS
    AbsSet
CONCRETE_VARIABLES
    var1,
    var2,
    var3
INVARIANT
    var1 ∈ AbsSet ∧
    var2 ∈ AbsSet ∧
    var3 ∈ AbsSet
...
END
```

```
IMPLEMENTATION
    MA_i
REFINES
    MA
VALUES
    AbsSet = 0 .. 100
INITIALISATION
    var1 := 0 ;
    var2 := 100 ;
    var3 := (var1 + var2) / 2
...
END
```

Dans l'exemple ci-dessus, les variables concrètes *var1*, *var2* et *var3*, définies dans la machine *MA*, sont du type *AbsSet*. Dans l'implantation *MA_i*, comme *AbsSet* est valué par un intervalle entier, le type des variables *var1*, *var2* et *var3* devient $\mathbb{Z}$. Elles peuvent donc désormais être initialisées à l'aide d'expressions arithmétiques.

Exemple de valuation d'ensemble

```
MACHINE
    MA
SETS
    EnsAbs1 ;
    EnsAbs2 ;
    EnsAbs3 ;
    EnsAbs4 ;
    EnsAbs5
...
END
```

```
IMPLEMENTATION
    MA_i
REFINES
    MA
SEES
    MB
VALUES
    EnsAbs1 = 0 .. 2 × c1 + 1 ;
    EnsAbs2 = Interv ;
    EnsAbs3 = IntervAbs ;
    EnsAbs4 = EnsAbs6
...
END
```

```
MACHINE
    MB
SETS
    EnsAbs5,
    EnsAbs6
CONCRETE_CONSTANTS
    c1,
    Interv,
    IntervAbs
PROPERTIES
    c1 ∈ 0 .. 100 ∧
    Interv ⊆ NAT ∧
    IntervAbs ⊆ EnsAbs6
END
```

Dans l'exemple ci-dessus, l'ensemble abstrait *EnsAbs1* est valué par un intervalle entier. *EnsAbs2* est valué par une constante intervalle entier de la machine *vue MB*. *EnsAbs3* est valué par une constante intervalle de type abstrait *EnsAbs6* provenant de *MB*. *EnsAbs4* est valué par l'ensemble abstrait *EnsAbs6* de *MB*. Enfin, *EnsAbs5* est valué implicitement par l'ensemble abstrait homonyme de *MB*.

Valuation des constantes concrètes scalaires

Chaque constante concrète scalaire est valuée en fonction de son type. Si elle appartient à $\mathbb{Z}$, elle doit être valuée par une expression arithmétique. Il faut alors prouver que chaque sous-expression arithmétique est bien définie et qu'elle est contenue dans l'ensemble des entiers concrets INT (cf. §7.25.2 *Les termes*). Si la constante concrète appartient à BOOL, elle doit être valuée par une expression booléenne. Si elle est de type énuméré ou abstrait, elle doit être valuée par une constante énumérée ou abstraite.

Dans le cas des constantes appartenant à un ensemble abstrait qui a déjà été valué dans la clause VALUES, on rappelle que le type de la constante est modifié ; il s'agit du type de l'ensemble valuant l'ensemble abstrait.

Exemple

Dans l'exemple ci-dessous, la constante concrète *c1* est valuée par un entier littéral. La constante *c2* est valuée par une expression arithmétique, à l'aide de la constante concrète *CteB2* de la machine *vue MB*. La constante *c3* est valuée par un élément énuméré de l'ensemble énuméré *COUL* déclaré dans *MB*. La constante *c4* est valuée légitimement par un entier littéral puisqu'elle est de type entier depuis la valuation de *EnsAbs1* par l'intervalle entier 0 .. 512. La constante *c5* est valuée par la constante concrète *cteB1* de *MB*. Enfin, la constante *c6* est valuée par une expression arithmétique à l'aide de la constante *c2* valuée précédemment.

```
MACHINE
    MA
SEES
    MB
SETS
    EnsAbs1
CONCRETE_CONSTANTS
    c1, c2, c3, c4, c5, c6
PROPERTIES
    c1 ∈ INT  ∧
    c2 ∈ NAT  ∧
    c3 ∈ COUL  ∧
    c4 ∈ EnsAbs1  ∧
    c5 ∈ EnsAbs2  ∧
    c6 ∈ NAT
…
END
```

```
MACHINE
    MB
SETS
    COUL = { Rouge, Vert, Bleu } ;
    EnsAbs2
CONCRETE_CONSTANTS
    cteB1,
    cteB2
PROPERTIES
    cteB1 ∈ EnsAbs2  ∧
    cteB2 ∈ - 10 .. 10
END
```

```
IMPLEMENTATION
    MA_i
REFINES
    MA
SEES
    MB
VALUES
    c1 = - 100  ;
    c2 = cteB2² + 4  ;
    c3 = Bleu ;
    EnsAbs1 = 0 .. 512  ;
    c4 = 42  ;
    c5 = cteB1  ;
    c6 = c2 + 1
...
END
```

Valuation des constantes concrètes tableaux

Une constante concrète tableau peut être valuée de trois manières différentes :

- par une constante concrète tableau d'une machine *vue* ou *importée*.
- par un ensemble de maplets. Un maplet permet de représenter un n-uplet dont les *n*-1 premiers éléments désignent les indices de l'élément du tableau et dont le dernier élément désigne la valeur de l'élément du tableau. Les indices d'un maplet doivent être des scalaires littéraux, alors que la valeur d'un maplet doit être une valeur scalaire quelconque.
- par un tableau dont tous les éléments ont la même valeur. Ce tableau est exprimé sous la forme du produit cartésien entre les ensembles indices du tableau et un singleton contenant la valeur à donner à tous les éléments du tableau.

Dans la valuation d'un tableau dont les éléments sont des entiers, si une valeur du tableau est définie par une expression arithmétique, il faut prouver que chaque sous-expression appartient à INT (cf. §7.25.2 *Les termes*).

Exemple

Dans l'exemple ci-dessous, la constante *tab1* est valuée à l'aide de la constante *tab4* de la machine *vue MB*. La constante *tab2* est valuée par un ensemble de maplets : l'élément d'indice (1) possède la valeur FALSE, (2) possède la valeur TRUE. La constante *tab3* est valuée par le tableau *EnsAbs* × BOOL × {0} qui à tout élément de l'ensemble de départ du tableau associe la valeur 0.

```
MACHINE
    MA
SEES
    MB
CONCRETE_CONSTANTS
    tab1, tab2, tab3
PROPERTIES
    tab1  ∈  (0 .. 2) × BOOL → EnsAbs  ∧
    tab2  ∈  (1 .. 2) ⤖ BOOL  ∧
    tab3  ∈  EnsAbs × BOOL → INT
...
END
```

```
MACHINE
    MB
SETS
    EnsAbs
CONCRETE_CONSTANTS
    tab4
PROPERTIES
    tab4 ∈ (0 .. 2) × BOOL → EnsAbs
...
END
```

```
IMPLEMENTATION
    MA_i
REFINES
    MA
SEES
    MB
VALUES
    tab1  =  tab4 ;
    tab2  =  { 1 ↦ FALSE, 2 ↦ TRUE }  ;
    tab3  =  EnsAbs × BOOL × {0}
...
END
```

Valuation des constantes concrètes intervalles

Chaque constante concrète intervalle est valuée en fonction de son type. Si elle est de type entier, elle peut être valuée par une constante intervalle entière ou par un intervalle dont les bornes sont des expressions arithmétiques. Si elle est de type abstrait, elle doit être valuée par une constante intervalle de type abstrait ou par un ensemble abstrait.

Dans la valuation d'un intervalle entier à l'aide d'expressions arithmétiques, il faut prouver que chaque sous-expression est bien définie et appartient à INT (cf. §7.25.2 *Les termes*).

Exemple

Dans l'exemple ci-dessous, la constante intervalle *c1* est valuée à l'aide de la constante *cteInterv1* de la machine *vue MB*. La constante intervalle *c2* est valuée par l'intervalle entier 0 .. MAXINT / 2 - 1, dont la borne supérieure est une expression arithmétique. L'ensemble abstrait *EnsAbs1* est valué par l'intervalle entier 0 .. 100. Par conséquent, les données dont le type était *EnsAbs1* ont désormais le type ℤ. La constante intervalle *c3* incluse dans *EnsAbs1* est valuée par l'intervalle 1 .. 6. La constante intervalle *c4* incluse dans l'ensemble abstrait *EnsAbs2* est valuée par la constante intervalle *cteInterv2* de la machine *vue MB*, cette dernière constante étant bien incluse dans *EnsAbs2*.

```
MACHINE
    MA
SEES
    MB
SETS
    EnsAbs1
CONCRETE_CONSTANTS
    c1, c2, c3, c4
PROPERTIES
    c1 ⊆ INT  ∧
    c2 = 0 .. (MAXINT / 2 - 1)  ∧
    c3 ⊆ EnsAbs1  ∧
    c4 ⊂ EnsAbs2
...
END
```

```
MACHINE
    MB
SETS
    EnsAbs2
CONCRETE_CONSTANTS
    cteInterv1, cteInterv2
PROPERTIES
    cteInterv1 ⊆ INT  ∧
    cteInterv2 ⊂ EnsAbs2
...
END
```

```
IMPLEMENTATION
    MA_i
REFINES
    MA
SEES
    MB
VALUES
    c1  =  cteInterv1   ;
    c2  =  0 .. ( MAXINT / 2 - 1 )   ;
    EnsAbs1  =  0 .. 100   ;
    c3  =  1 .. 6   ;
    c4  =  cteInterv2
...
END
```

Visibilité

Dans la clause VALUES d'une implantation, les constantes concrètes et les ensembles énumérés des machines *vues* et *importées* sont accessibles en lecture. Les constantes concrètes et les ensembles abstraits de l'implantation ne peuvent se présenter qu'en partie gauche des valuations, et ils ne sont donc pas accessibles en lecture.

7.18 La clause CONCRETE_VARIABLES

Syntaxe

Clause_concrete_variables ::= "CONCRETE_VARIABLES" *Ident_ren*$^{+","}$

Description

La clause CONCRETE_VARIABLES définit la liste des variables concrètes d'un composant.

Une variable concrète est une donnée implémentable dans un langage informatique. Elle peut être un scalaire ou un tableau (cf. §3.4 *Types et contraintes des données concrètes*). Une variable concrète n'a pas à être raffinée puisqu'elle est implicitement conservée au cours du raffinement jusqu'à l'implantation. Cette propriété autorise alors l'accès en lecture directe aux variables concrètes d'une machine dans les implantations qui *importent* cette machine, sans recourir nécessairement à un service de lecture comme c'est nécessairement le cas pour les variables abstraites.

Restrictions

1. Les variables concrètes déclarées dans une machine abstraite ne doivent pas être renommées.
2. Le nom d'une nouvelle variable concrète d'un raffinement ou d'une implantation doit différer du nom des variables (concrètes ou abstraites) de l'abstraction, sauf dans le cas suivant : une variable concrète peut raffiner une variable abstraite homonyme de l'abstraction, elle est alors implicitement égale à la variable abstraite.

Utilisation

Les variables concrètes d'une machine abstraite regroupent les variables concrètes définies dans la machine et provenant des *inclusions* (cf. §7.9 *La clause INCLUDES*). Les variables concrètes d'un raffinement regroupent les variables concrètes définies dans le raffinement, provenant de l'abstraction et provenant de machines *incluses*. Les variables concrètes d'une implantation regroupent les variables concrètes définies dans l'implantation et provenant de l'abstraction.

Le typage et les propriétés invariantes des variables concrètes sont exprimés dans la clause INVARIANT (cf. §7.20 *La clause INVARIANT*).

Chaque variable concrète appartenant à un composant doit être initialisée dans la clause INITIALISATION du composant (cf. §7.22 *La clause INITIALISATION*).

Chaque variable concrète définie dans une machine abstraite doit être typée dans la clause INVARIANT de la machine. Une variable concrète définie dans un raffinement ou dans une implantation peut être :

- soit une nouvelle variable concrète. Elle doit alors être typée explicitement dans la clause INVARIANT du raffinement ou de l'implantation.
- soit le raffinement d'une variable abstraite, si son nom est identique à celui d'une variable abstraite de l'abstraction. Pour cela, la variable abstraite doit être de même nature qu'une variable concrète (scalaire ou tableau). Elle est alors typée implicitement par un prédicat de liaison qui signifie que la variable concrète est égale à la variable abstraite homonyme.

Visibilité

Les variables concrètes d'un composant sont utilisables dans les clauses INVARIANT et ASSERTIONS de ce composant et de ses raffinements successifs. Elles sont accessibles en lecture et en écriture dans le corps de l'initialisation et des opérations du composant. Les variables concrètes déclarées dans une machine sont accessibles en lecture uniquement par les composants qui *voient*, *incluent*, *utilisent* ou *importent* cette machine.

Exemple

```
MACHINE
    MA
CONCRETE_VARIABLES
    Production,
    ProductionMois
ABSTRACT_VARIABLES
    ProductionExterne
INVARIANT
    Production ∈ INT ∧
    ProductionMois ∈ (1 .. 12) → INT ∧
    ProductionExterne ∈ INT
INITIALISATION
    Production :∈ INT ||
    ProductionMois :∈ (1 .. 12) → INT ||
    ProductionExterne :∈ INT
...
END
```

```
IMPLEMENTATION
    MA_i
REFINES
    MA
CONCRETE_VARIABLES
    Recettes,
    Charges,
    ProductionExterne
INVARIANT
    Recettes ∈ INT ∧
    Charges ∈ INT
INITIALISATION
    Production := 0 ;
    ProductionMois := (1 .. 12) × {0} ;
    ProductionExterne := 0 ;
    Recettes := 0 ;
    Charges := 0
...
END
```

La machine abstraite *MA* définit deux variables concrètes *Production* et *ProductionMois* et une variable abstraite *ProductionExterne*. Les variables *Production* et *ProductionExterne* sont des entiers concrets et la variable *ProductionMois* est un tableau d'entiers concrets d'indice l'intervalle 1..12. Toutes ces variables sont initialisées dans la clause INITIALISATION. L'implantation *MA_i* de *MA* définit les nouvelles variables concrètes *Recettes*, *Charges* et *ProductionExterne*. Cette dernière raffine la variable abstraite homonyme de *MA*. Toutes les variables de *MA_i* (*Production*, *ProductionMois*, *ProductionExterne*, *Recettes* et *Charges*) sont initialisées dans la clause INITIALISATION.

7.19 La clause ABSTRACT_VARIABLES

Syntaxe

Clause_abstract_variables ::=
"ABSTRACT_VARIABLES" *Ident_ren*$^{+","}$
| "VARIABLES" *Ident_ren*$^{+","}$

Machine abstraite

La clause ABSTRACT_VARIABLES permet d'introduire dans une machine ou un raffinement de nouvelles variables abstraites. Une variable abstraite est une donnée dont le type est quelconque et qui est raffinée au cours du raffinement du composant.

Restrictions

1. Les variables abstraites déclarées dans une machine ne doivent pas être renommées.
2. Le nom d'une nouvelle variable abstraite d'un raffinement doit différer du nom des variables (concrètes ou abstraites) de l'abstraction, sauf dans le cas suivant : une variable abstraite peut raffiner une variable abstraite homonyme de l'abstraction, elle est alors implicitement égale à la variable abstraite.

Utilisation

Le nom de la clause est suivi par une liste d'identificateurs qui représentent le nom des variables abstraites. Les variables abstraites doivent être typées (cf. §3.3 *Typage des données abstraites*) et éventuellement contraintes dans la clause INVARIANT. Elles doivent toutes être initialisées dans la clause INITIALISATION.

Dans un raffinement, chaque variable abstraite définie peut être :

- soit une nouvelle variable abstraite. Son nom doit alors être un identificateur non renommé. Elle doit être typée et éventuellement contrainte dans la clause INVARIANT.
- soit le raffinement d'une variable abstraite, si son nom est identique à celui d'une variable abstraite du composant raffiné. L'identificateur de la variable est renommé si la variable du composant raffiné provient d'une machine *incluse* (ou transitivement *incluse*) renommée. Il est alors inutile de typer la variable abstraite dans la clause INVARIANT, car elle est typée par défaut à l'aide d'un invariant de liaison implicite qui signifie que la nouvelle variable abstraite est égale à la variable abstraite homonyme du composant raffiné. D'autres propriétés invariantes portant sur la variable abstraite peuvent également être exprimées dans la clause INVARIANT.

Si le raffinement M_n raffine M_{n-1}, les variables abstraites de M_{n-1} peuvent également être raffinées en tant que variables concrètes de M_n. Si une variable de M_{n-1} n'est pas raffinée en tant que variable abstraite, ni en tant que variable concrète, alors elle disparaît dans M_n. Elle ne fait donc plus partie des variables du composant M_n.

Les variables abstraites du raffinement qui ne proviennent pas d'une instance de machine *incluse* par le raffinement doivent être initialisées dans la clause INITIALISATION.

Dans une implantation, il est interdit de définir des variables abstraites.

Visibilité

Les variables abstraites d'un composant sont accessibles dans les clauses INVARIANT et ASSERTIONS de ce composant. Elles sont accessibles en lecture et en écriture dans le corps de l'initialisation et des opérations du composant. Elles sont accessibles en lecture par les composants qui *importent*, *voient*, *incluent* ou *utilisent* cette machine (cf. Annexe C. *Tables de visibilité*).

Soient M_A et M_B des machines. Si M_A *voit* M_B, les variables abstraites de M_B sont accessibles dans M_A en lecture dans le corps de l'initialisation et des opérations de M_A. Si M_A *utilise*, ou *inclut* M_B, les variables abstraites de M_B sont accessibles dans M_A dans les clauses INVARIANT et ASSERTIONS et elles sont accessibles en lecture dans le corps de l'initialisation et des opérations.

Si M_n raffine M_{n-1}, les variables abstraites de M_{n-1} qui disparaissent dans M_n sont seulement accessibles dans M_n, dans les clauses INVARIANT et ASSERTIONS ainsi que dans les clauses INITIALISATION et OPERATIONS, au sein des prédicats des substitutions assertion et des variants et invariants de boucles. Elles ne sont plus accessibles dans les raffinements successifs de M_n.

Soient M_A un raffinement et M_B une machine. Si M_A *voit* (clause SEES) M_B, les variables abstraites de M_B sont accessibles dans M_A en lecture dans le corps de l'initialisation et des opérations.

Si M_A *inclut* M_B, les variables abstraites de M_B sont accessibles dans M_A dans les clauses INVARIANT et ASSERTIONS et elles sont accessibles en lecture dans le corps de l'initialisation et des opérations.

Les variables abstraites déclarées dans un raffinement ne sont pas accessibles par les composants externes au module.

7.20 La clause INVARIANT

Syntaxe

Clause_invariant ::= "INVARIANT" *Prédicat*

Description

La clause INVARIANT contient, au sein d'un prédicat appelé invariant, d'une part le typage des variables déclarées dans le composant et d'autre part des propriétés sur ces variables.

L'invariant exprime les propriétés invariantes des variables de la machine abstraite. Il faut prouver que l'initialisation du composant (cf. §7.22 *La clause INITIALISATION*) établit l'invariant et que chaque appel d'une opération de la machine préserve l'invariant. Des composants extérieurs peuvent accéder aux variables concrètes d'une machine abstraite en lecture, mais pas en écriture, afin de ne pas briser l'invariant.

L'invariant d'un raffinement ou d'une implantation permet d'exprimer le lien entre les nouvelles variables du composant et les variables de l'abstraction.

Restrictions

1. Chaque variable, concrète ou abstraite, définie dans un composant et qui n'est pas homonyme à une variable de l'éventuelle abstraction du composant, doit être typée, avant toute autre utilisation, dans la clause INVARIANT du composant par un prédicat de typage (cf. §3.6 *Typage des variables concrètes* et §3.3 *Typage des données abstraites*) situé au plus haut niveau dans une série de conjonctions. Ces variables ne peuvent pas être utilisées dans l'invariant avant d'avoir été typées.
2. Chaque variable définie dans un raffinement ou une implantation et qui est homonyme à une variable de l'abstraction ne doit pas être typée, puisqu'elle est typée implicitement par un prédicat qui signifie que la nouvelle variable est égale à la variable homonyme de l'abstraction. Les deux variables doivent alors être de même type.
3. Si une variable d'une instance de machine *incluse* par un raffinement est homonyme et de même nature (abstraite ou concrète) à une variable de l'abstraction du raffinement, alors les deux variable homonymes désignent la même donnée et elles doivent être de même type.
4. Si une variable d'une instance de machine *importée* par une implantation est homonyme et de même nature (abstraite ou concrète) à une variable de l'abstraction de l'implantation, alors les deux variable homonymes désignent la même donnée et elles doivent être de même type.

Utilisation

Le nom de la clause INVARIANT est suivi d'une liste de prédicats. Dans une machine abstraite l'invariant permet de typer les variables définies dans la machine et de définir leurs propriétés invariantes.

L'invariant d'un raffinement permet de typer les nouvelles variables du raffinement et de définir leurs propriétés, notamment les liens existant entre les variables du raffinement et celles de l'abstraction.

L'invariant d'une implantation est semblable à celui d'un raffinement. Il permet également de définir la relation d'implantation entre, d'une part, les variables de l'implantation et d'autre part, les variables des instances de machines *importées* par l'implantation.

Typage des variables

Les variables doivent être typées dans l'un des prédicats situés au plus haut niveau d'analyse syntaxique de la clause INVARIANT séparés par des conjonctions '∧', à l'aide de prédicats de typage de données abstraites pour les variables abstraites (cf. §3.3 *Typage des données abstraites*) et de prédicat de typage de variables concrètes pour les variables concrètes (cf. §3.6 *Typage des variables concrètes*).

Les variables déclarées dans le raffinement peuvent se scinder en deux groupes : les nouvelles variables et les variables qui raffinent des variables abstraites homonymes de l'abstraction. Chaque nouvelle variable doit être typée par un invariant de typage. Il est inutile de typer une variable qui raffine une variable abstraite homonyme de l'abstraction, car elle est typée par défaut à l'aide d'un invariant de liaison implicite qui signifie que la nouvelle variable est égale à la variable abstraite de l'abstraction.

Comme les nouvelles variables déclarées dans une implantation ne peuvent être que des variables concrètes, le typage des variables dans l'invariant concerne seulement les nouvelles variables concrètes déclarées dans l'implantation.

Liaison entre variables

L'invariant d'un raffinement permet de préciser quels liens unissent les variables déclarées dans le raffinement et les variables de son abstraction. Chaque prédicat de l'invariant définissant un tel lien est appelé invariant de liaison. Un invariant de liaison peut aussi bien prendre la forme d'un prédicat de typage que d'une propriété. On rappelle également qu'un invariant de liaison implicite lie par défaut deux variables homonymes, l'une étant déclarée dans le raffinement, l'autre dans son abstraction.

Les variables abstraites de l'abstraction de l'implantation peuvent être liées aux variables concrètes ou abstraites des instances de machines *importées* (voir l'exemple ci-dessous).

Une variable d'un raffinement peut être homonyme à une variable de même nature concrète ou abstraite d'une instance de machine *incluse* par le raffinement. Deux variables homonymes fusionnent, elles représentent donc la même donnée. Elles doivent alors être de même type.

Soit *M_i* une implantation, si une variable concrète de l'abstraction de *M_i* porte le même nom qu'une variable concrète d'une instance de machine *importée* *Mimp*, alors les deux variables sont automatiquement liées par un invariant de liaison implicite qui signifie que les variables sont égales. Par conséquent, les deux variables doivent avoir le même type. Les deux variables fusionnent ; on dit que la variable de *M_i* s'implante sur la variable homonyme de *Mimp*. La variable concrète de *M_i* se comporte alors comme une référence sur la variable homonyme de *Mimp*. L'instance de machine *Mimp* devient responsable de la gestion de la variable concrète et notamment de son initialisation.

Il est possible d'implanter une variable concrète de l'implantation *M_i* par une variable concrète portant un nom différent d'une instance de machine *importée* *Mimp* en écrivant en invariant l'égalité des deux variables. Cependant ce cas est totalement dénué d'intérêt. En effet, les deux variables ne fusionnent pas. Pour ne pas briser l'invariant, il faut donc modifier conjointement les deux variables.

Exemple

Dans l'exemple ci-dessous, la variable abstraite *var1* de la machine *MA* est implantée par la variable *vimp*, la variable abstraite *var2* est implantée implicitement par homonymie avec la variable *var2* de la machine importée *Mimp*, et la variable concrète *var3* est implantée dans l'implantation. La variable concrète *var5* est implantée par homonymie avec la variable *var5* de la machine importée *Mimp*.

```
MACHINE
    MA
ABSTRACT_VARIABLES
    var1,
    var2
CONCRETE_VARIABLES
    var5
INVARIANT
    var1 ∈ NAT ∧
    var2 ∈ BOOL ∧
    var5 ∈ INT ∧
    (var1 > var5 ∧ var2 = TRUE)
...
END
```

```
IMPLEMENTATION
    MA_i
REFINES
    MA
IMPORTS
    Mimp
CONCRETE_VARIABLES
    var3
INVARIANT
    var3 ∈ NAT ∧
    var1 = vimp
...
END
```

```
MACHINE
    Mimp
ABSTRACT_VARIABLES
    vimp,
    var2
CONCRETE_VARIABLES
    var5
INVARIANT
    vimp ∈ 1 .. 100 ∧
    var2 ∈ BOOL ∧
    var5 ∈ INT
...
END
```

Visibilité

Dans la clause INVARIANT d'une machine, les paramètres formels, les ensembles (abstraits et énumérés), les constantes et les variables de la machine sont accessibles. Les ensembles, les constantes, les éléments d'ensembles énumérés et les variables des machines *incluses* sont accessibles. Les paramètres, les ensembles, les constantes, les éléments d'ensembles énumérés et les variables des machines *utilisées* sont accessibles. Les ensembles, les constantes et les éléments d'ensembles énumérés des machines *vues* sont accessibles.

Dans la clause INVARIANT d'un raffinement, les paramètres formels, les ensembles (abstraits et énumérés), les constantes, les éléments d'ensembles énumérés et les variables du raffinement sont accessibles. Les constantes et les variables de l'abstraction disparaissant dans le raffinement sont accessibles. Les ensembles, les constantes, les éléments d'ensembles énumérés et les variables des machines *incluses* sont accessibles. Les ensembles, les constantes et les éléments d'ensembles énumérés des machines *vues* sont accessibles.

Dans l'invariant d'une implantation, les constituants suivants sont accessibles :

- les paramètres formels, les ensembles, les éléments énumérés, les constantes concrètes et les variables concrètes de l'implantation,
- les constantes abstraites et les variables abstraites de l'abstraction de l'implantation,
- les ensembles, les éléments énumérés, les constantes et les variables des instances de machines *vues* ou *importées* par l'implantation.

Exemple

```
MACHINE
    MA
CONCRETE_VARIABLES
    var1
ABSTRACT_VARIABLES
    var2
INVARIANT
    var1 ∈ INT ∧
    var2 ∈ ℕ ∧
    (var1 ≤ 0 ∧ var1 + var2 = 0)
INITIALISATION
    var1 :∈ MININT .. 0 ||
    var2 := - var1
...
END
```

7.21 La clause ASSERTIONS

Syntaxe

Clause_assertions ::= "ASSERTIONS" *Prédicat*$^{+";"}$

Description

La clause ASSERTIONS comporte une liste de prédicats appelés assertions portant sur les variables du composant. Ces assertions sont des résultats intermédiaires déduits de l'invariant du composant susceptibles d'aider la preuve du composant.

Une assertion est un lemme qui devra être prouvé à partir de l'invariant du composant et des lemmes séparés par des ';' qui le précèdent dans la clause ASSERTIONS. L'ordre des assertions est donc significatif. Dans les obligations de preuve concernant les opérations du composant, les assertions sont ajoutées en hypothèse, en plus de l'invariant.

Exemple

Dans l'exemple ci-dessous, la variable concrète *var* est un entier implémentable qui vérifie $var^2 = 1$. Nous ajoutons une assertion pour dire que la variable *var* est égale à 1 ou à - 1. En effet, cette assertion peut être prouvée en prenant comme hypothèse l'invariant. Lors de la construction des autres Obligations de Preuve de la machine, chaque fois que l'invariant apparaît en hypothèse alors cette assertion sera ajoutée afin de faciliter la démonstration de l'Obligation de Preuve.

```
MACHINE
    MA
CONCRETE_VARIABLES
    var
INVARIANT
    var ∈ INT ∧
    var² = 1
ASSERTIONS
    var = 1 ∨ var = - 1
...
END
```

7.22 La clause INITIALISATION

Syntaxe

Clause_initialisation ::= "INITIALISATION" *Substitution*

Clause_initialisation_B0 ::= "INITIALISATION" *Instruction*

Description

La clause INITIALISATION permet d'initialiser toutes les variables du composant. Il faut prouver que l'initialisation d'un composant établit l'invariant.

La clause INITIALISATION peut être considérée comme la déclaration d'une opération particulière d'un module. Cette opération ne possède pas de paramètre. Son rôle est d'initialiser les variables du module afin qu'elles établissent l'invariant du composant. Lors de l'exécution du code B0 associé à un projet, toutes les opérations d'initialisation des modules sont appelées dans un ordre de dépendance correct, avant le lancement du point d'entrée du projet.

Restriction

1. Toutes les variables du composant non homonymes à des variables d'instances de machines *incluses* ou *importées* doivent être initialisées dans la clause INITIALISATION.

Utilisation

Le nom de la clause est suivi par une substitution de spécification. L'ensemble de ces substitutions est décrit dans le chapitre 6 *Substitution*.

Dans une machine abstraite, toutes les variables du composant doivent être initialisées dans l'initialisation. Les variables des machines *incluses* par le composant ne doivent pas être initialisées par le composant puisqu'elles sont déjà initialisées dans la clause INITIALISATION de leur machine.

Lors de l'initialisation des variables d'un composant, les variables des machines dont il dépend (*incluses*, *importées*, *vues* ou *utilisées*) sont considérées comme déjà initialisées. L'initialisation permet de modifier les variables des machines *incluses* ou *importées* en appelant des opérations de ces machines. Cette possibilité est notamment utile pour établir les invariants qui expriment des propriétés de liaison sur des variables de machines *incluses* ou *importées*.

Dans un raffinement, les substitutions de l'initialisation doivent être des substitutions de raffinement. Comme dans les machines, toutes les variables du raffinement qui ne proviennent pas d'une *inclusion* d'instance de machine par le raffinement doivent être initialisées. Il s'agit des variables concrètes des abstractions du raffinement et des nouvelles variables, déclarées dans le raffinement.

Dans une implantation, la clause INITIALISATION permet d'initialiser les variables concrètes de l'implantation. Les substitutions utilisées dans l'initialisation doivent cependant être des substitutions d'implantation encore appelées instructions (cf. §7.25.4 *Les instructions*). Toutes les variables concrètes de l'implantation doivent être initialisées. Il existe deux manières d'initialiser une variable concrète d'une implantation *M_i* :

- directement, en donnant explicitement une valeur à la variable concrète au sein d’une instruction de l’initialisation. La variable concrète est alors localisée dans *M_i*.
- indirectement, si la variable concrète porte le même nom qu’une variable concrète d’une instance de machine importée *Mimp*. Alors les deux variables homonymes doivent avoir le même type. La variable de *M_i* ne doit pas être initialisée explicitement dans la clause INITIALISATION de *M_i*. On dit qu’elle est délocalisée dans *Mimp*. La variable concrète de *M_i* ne représente alors qu’une référence sur la variable homonyme de *Mimp*. Par conséquent, l’initialisation effective de la variable concrète est déportée dans la machine *Mimp*.

Bien qu’il soit possible d’utiliser dans l’initialisation toutes les sortes d’instructions, la manière usuelle d’initialiser les variables consiste à utiliser des substitutions « devient égal » ou des appels d’opérations.

Visibilité

Les variables d'une machine sont accessibles en lecture et en écriture dans la clause INITIALISATION de la machine. Les paramètres, les ensembles et les constantes de la machine sont accessibles en lecture. Les ensembles, les constantes, les éléments d’ensembles énumérés et les variables des instances de machines *incluses*, *utilisées* ou *vues* sont accessibles en lecture. En outre, les paramètres des machines *utilisées* sont accessibles en lecture.

Dans l’initialisation d’une machine M_1, il est possible d’accéder aux opérations des machines *incluses* par M_1 et aux opérations de consultation des machines *vues* par M_1. Par contre, il est interdit d’accéder aux opérations propres de M_1 et aux opérations des machines *utilisées* par M_1.

Les variables d’un raffinement sont accessibles en écriture dans la clause INITIALISATION du raffinement. Les paramètres, les ensembles, les constantes et les éléments d’ensembles énumérés du raffinement sont accessibles en lecture dans l’initialisation. Les ensembles, les constantes et les variables des machines *incluses* ou *vues* par le raffinement sont accessibles en lecture dans l’initialisation.

Dans l’initialisation d’un raffinement M_1, il est possible d’accéder aux opérations des machines *incluses* par M_1 et aux opérations de consultation des machines *vues* par M_1. Par contre, il est interdit d’accéder aux opérations propres de M_1.

Les variables d'une implantation sont accessibles en lecture et en écriture dans les instructions de la clause INITIALISATION de l’implantation. Elles doivent être écrites avant d’être lues. Les paramètres formels, les éléments énumérés et les constantes de l’implantation sont accessibles en lecture dans l’initialisation. Les éléments énumérés, les constantes concrètes et les variables concrètes des machines *vues* ou *importées* par l’implantation sont accessibles en lecture dans l’initialisation. Les paramètres formels de l’implantation, les ensembles, les constantes, les éléments d’ensembles énumérés et les variables de l’implantation et des instances de machines *vues* ou *importées* sont accessibles dans les invariants de boucles WHILE de l’initialisation et dans les prédicats des substitutions ASSERT.

Exemple

Dans l'exemple ci-dessous, la variable abstraite *var1* de la machine *MA* est implantée par la variable *vimp*, la variable *var2* est implantée implicitement par homonymie avec la variable *var2* de la machine importée *Mimp*, et les variables concrètes *var3* et *var4* sont implantées localement comme des variables propres.

```
MACHINE
    MA
ABSTRACT_VARIABLES
    var1,
    var2
CONCRETE_VARIABLES
    var4
INVARIANT
    var1 ∈ NAT ∧
    var2 ∈ BOOL ∧
    var4 ∈ INT ∧
...
END
```

```
IMPLEMENTATION
    MA_i
REFINES
    MA
IMPORTS
    Mimp
CONCRETE_VARIABLES
    var3
INVARIANT
    var3 ∈ NAT ∧
    var1 = vimp
INITIALISATION
    var3 := 0 ;
    setv2 ( TRUE ) ;
    setvimp (10) ;
    var4 := - 12
...
END
```

```
MACHINE
    Mimp
CONCRETE_VARIABLES
    vimp,
    var2
INVARIANT
    vimp ∈ 1 .. 100 ∧
    var2 ∈ BOOL
INITIALISATION
    vimp :∈ 1 .. 100 ||
    var2 :∈ BOOL
OPERATIONS
    setv2 (b0) =
    PRE
        b0 ∈ BOOL
    THEN
        var2 := b0
    END ;
    setvimp (in) =
    PRE
        in ∈ 1 .. 100
    THEN
        vimp := in
    END
END
```

7.23 La clause OPERATIONS

Syntaxe

Clause_operations ::= "OPERATIONS" *Opération*$^{+";"}$

Opération ::= *Entête_opération* "=" *Substitution_corps_opération*

Entête_opération ::= [Ident$^{+","}$ "←"] *Ident_ren* ["(" Ident$^{+","}$ ")"]

Clause_operations_B0 ::= "OPERATIONS" *Opération_B0*$^{+";"}$

Opération_B0 ::= *Entête_opération* "=" *Instruction_corps_opération*

Substitution_corps_opération ::=
Substitution_bloc
| *Substitution_identité*
| *Substitution_devient_égal*
| *Substitution_précondition*
| *Substitution_assertion*
| *Substitution_choix_borné*
| *Substitution_conditionnelle*
| *Substitution_sélection*
| *Substitution_cas*
| *Substitution_any*
| *Substitution_let*
| *Substitution_devient_elt_de*
| *Substitution_devient_tel_que*
| *Substitution_variable_locale*
| *Substitution_appel_opération*

Instruction_corps_opération ::=
Instruction_bloc
| *Instruction_variable_locale*
| *Substitution_identité*
| *Instruction_devient_égal*
| *Instruction_appel_opération*
| *Instruction_conditionnelle*
| *Instruction_cas*
| *Instruction_assertion*
| *Substitution_tant_que*

Restrictions

1. Les paramètres formels d'une opération doivent être deux à deux distincts.
2. Dans une machine abstraite, les opérations déclarées dans la clause opération ne doivent pas être renommées.
3. Dans une machine abstraite, les paramètres d'entrée d'une opération doivent être typés dans le prédicat de la substitution précondition qui doit débuter le corps de l'opération, par des prédicats de typage (cf. §3.7 *Typage des paramètres d'entrée d'opération* et §3.3 *Typage des données abstraites*) situés au plus haut niveau d'analyse syntaxique dans une série de conjonctions. Ces paramètres d'entrée ne peuvent pas être utilisés dans le prédicat de la substitution précondition avant d'avoir été typés.
4. Dans une machine abstraite, les paramètres de sortie d'une opération doivent être typés dans le corps de l'opération par des substitutions de typage (cf. §3.9 *Typage des variables locales et des paramètres de sortie d'opération*). Ces paramètres de sortie ne peuvent pas être utilisés dans le corps de l'opération avant d'avoir été

typés.

5. Dans un raffinement d’une machine abstraite, on ne doit pas définir de nouvelles opérations.
6. Dans une implantation, chaque opération de la machine abstraite doit être déclarée, soit en tant qu’opération propre, dans la clause OPERATIONS, soit en tant qu’opération promue, dans la clause PROMOTES (cf. §7.10 *La clause PROMOTES*).
7. Chaque opération d'un raffinement ou d'une implantation doit avoir les mêmes paramètres formels et dans le même ordre que sa spécification (l’opération homonyme de la clause LOCAL_OPERATIONS dans le cas d’une opération locale d’implantation, l’opération homonyme de la machine abstraite sinon).
8. Dans la clause OPERATIONS d’une implantation, on ne peut définir que des opérations qui sont spécifiées dans la machine abstraite de l’implantation, ou dans la clause LOCAL_OPERATIONS.
9. Dans une implantation, chaque opération locale spécifiée dans la clause LOCAL_OPERATIONS doit être implantée dans la clause OPERATIONS.
10. Le graphe d’appel d’opérations locales, en ne considérant que les implémentations d’opérations locales de la clause OPERATIONS, ne doit pas contenir de cycle.

Description

La clause OPERATIONS permet de déclarer dans un composant des opérations. Les opérations constituent la partie dynamique du langage B puisqu'elles peuvent faire évoluer les variables d'un composant. Une opération peut posséder des paramètres d'entrée et des paramètres de sortie. Les opérations des machines abstraites constituent les spécifications de l'opération pour le module. Les opérations d'une machine sont utilisables par d'autres machines sous la forme d'appels d'opérations (cf. §6.16 *Substitution appel d’opération*).

La clause OPERATIONS permet également de déclarer l’implantation des opérations locales, spécifiées dans la clause LOCAL_OPERATIONS (cf. §7.24 *La clause LOCAL_OPERATIONS*).

Il faut démontrer que les opérations d'une machine abstraite préservent l'invariant de la machine. Les opérations doivent être raffinées jusqu'à l'implantation pour devenir des opérations informatiques. Il faut démontrer à chaque étape que l'opération est cohérente avec l'opération qu'elle raffine.

Utilisation dans une machine abstraite

L’ensemble des opérations d’une machine abstraite se compose des opérations promues (cf. §7.10 *La clause PROMOTES* et §7.11 *La clause EXTENDS*) et des opérations de la clause OPERATIONS. Ces dernières sont encore appelées opérations propres.

La clause OPERATIONS permet de déclarer les services offerts par une machine et de spécifier leur comportement. Les opérations d’une machine abstraite constituent la partie dynamique de la machine, par opposition aux données (les ensembles, les constantes, les variables et les paramètres des machines) qui constituent la partie statique. Elles permettent, en effet, de modifier les données de la machine. Il faudra prouver que l’appel d’une opération d’une machine préserve l’invariant de la machine.

Les paramètres d’opérations des modules développés (cf. §8.2 *Module B*) et des machines abstraites doivent être de type concret puisqu’ils seront associés à un code, alors que les paramètres d'opérations des modules abstraits peuvent être de type quelconque puisqu'ils ne servent que d'intermédiaire de raisonnement.

Une opération se compose d'un en-tête et d'un corps.

En-tête d'opération

L'en-tête d'une opération est constitué d'un identificateur désignant le nom de l'opération et des éventuels paramètres formels d'entrée et de sortie de l'opération. Le nom d'une opération propre déclarée dans une machine abstraite ne doit pas posséder de renommage. Les paramètres d'entrée sont représentés par une liste d'identificateurs parenthésée qui suit le nom de l'opération. Les paramètres de sortie sont représentés par une liste d'identificateurs précédant le nom de l'opération. Les paramètres d'entrée et de sortie d'une opération doivent être deux à deux distincts.

Passage des paramètres par valeur

En B, lors de l'appel d'une opération, la sémantique du passage des paramètres est la copie.

- Les paramètres d'entrée de l'opération permettent de paramétrer un appel d'opération à l'aide de valeurs. Lors d'un appel d'opération, la valeur de chaque paramètre effectif d'entrée est recopiée dans le paramètre formel.
- Les paramètres de sortie de l'opération permettent de renvoyer les résultats d'un appel d'opération sous la forme de valeurs. Après un appel d'opération, la valeur de chaque paramètre formel de sortie est recopiée dans le paramètre effectif.

Ces copies sont formalisées par des substitutions « devient égal » entre les paramètres formels et effectifs dans la définition de la substitution d'appel d'opération (cf. §6.16 *Substitution appel d'opération*).

Règles de portée

La portée des paramètres formels définis dans l'en-tête d'une opération est le corps de l'opération. Les paramètres formels d'entrée d'opération sont accessibles en lecture uniquement dans les prédicats, par exemple les prédicats des substitutions IF ou WHILE, et dans les substitutions. Les paramètres formels de sortie d'opération sont accessibles dans les prédicats en lecture et dans les substitutions en lecture et en écriture. Pour pouvoir lire un paramètre formel de sortie, il faut d'abord lui avoir donné une valeur.

Règles de typage

Les paramètres d'opérations des modules possédant un code associé (modules développés par raffinements successifs ou machines abstraites) doivent être de type implémentable. Les types permis sont ceux d'une variable concrète (type entier, booléen, ensemble abstrait, ensemble énuméré ou tableau). Dans le cas des paramètres d'entrée d'opération, on permet également le type chaîne de caractères, ce qui donne la possibilité d'envoyer un message à l'aide d'un appel d'opération.

Les paramètres d'opérations des modules abstraits peuvent être de type quelconque (cf. §3.3 *Typage des données abstraites*).

Les paramètres d'entrée d'opération

Les paramètres formels d'entrée doivent être typés dans le corps de l'opération, au sein d'un prédicat de typage. Pour pouvoir utiliser un paramètre formel d'entrée dans l'opération, il faut l'avoir typé dans le texte qui précède son utilisation. Une opération possédant des paramètres d'entrée s'écrit à l'aide d'une substitution précondition, qui type les paramètres formels d'entrée puis qui permet éventuellement d'exprimer d'autres propriétés que doivent vérifier ces paramètres d'entrée. Lors de la spécification de l'opération, on suppose que les paramètres formels d'entrée vérifient la précondition et lors d'un appel à cette opération, on doit prouver que les paramètres effectifs d'entrée vérifient la précondition. Les paramètres formels d'entrée ne peuvent pas être modifiés dans le corps de l'opération.

Exemple

```
MACHINE
    MA
OPERATIONS
    Service1 (x1, b1, tab1, mess) =
    PRE
        x1 ∈ NAT ∧
        b1 ∈ BOOL ∧
        tab1 ∈ (0 .. 10) × (0 .. 10) → INT ∧
        mess ∈ STRING ∧
        ...
    THEN
    ...
    END
END
```

Les paramètres de sortie d'opération

Les paramètres formels de sortie doivent être typés dans le corps de l'opération. Pour pouvoir utiliser un paramètre formel de sortie dans l'opération, il faut l'avoir typé dans le texte qui précède son utilisation. La manière usuelle d'écrire une opération possédant des paramètres de sortie consiste à les typer en leur donnant une valeur dans une substitution « devient égal », « devient appartient », « devient tel que » ou comme paramètre de sortie d'un appel d'opération.

Exemple

```
MACHINE
    MA
OPERATIONS
    ok, res1, tab2 ⟵ Service2 =
    BEGIN
        res1 : (res1 ∈ 0 .. 10  ∧  res1 / 2 = 0)  ||
        tab2 :∈ (0 .. 10) × (0 .. 10) → INT  ||
        ...
        ok := bool ( ... )
        ...
    END
END
```

Corps d'opération

Le corps d'une opération propre est constitué d'une substitution. Seules les substitutions de niveau spécification sont autorisées (cf. chapitre 6 *Substitutions*).

Exemple

```
MACHINE
    MA
OPERATIONS
    res_min, res_max, egal ⟵ Comparer (x1, x2) =
    PRE
        x1 ∈ INT  ∧
        x2 ∈ INT
    THEN
        res_min := min ({x1, x2})  ||
        res_max := max ({x1, x2})  ||
        egal := bool (x1  = x2)
    END
END
```

Visibilité

Les variables du composant sont accessibles en lecture et en écriture dans la clause OPERATIONS du composant. Les paramètres, les ensembles, les constantes et les éléments d'ensembles énumérés du composant sont accessibles en lecture. Les ensembles, les constantes, les éléments d'ensembles énumérés et les variables des machines *incluses*, *utilisées* ou *vues* sont accessibles en lecture. En outre, les paramètres des machines *utilisées* sont accessibles en lecture.

Dans le corps d'une opération propre d'une machine M_1, il est possible d'accéder aux opérations des machines *incluses* par M_1 et aux opérations de consultation des machines *vues* par M_1. Par contre, il est interdit d'accéder aux opérations propres de M_1 et aux opérations des machines *utilisées* par M_1.

Utilisation dans un raffinement

Raffinement d'une opération

Dans les raffinements successifs d'une machine, chaque opération de la machine, qu'elle soit propre ou promue, doit être raffinée par une opération. Il est interdit de déclarer de nouvelles opérations dans un raffinement.

Le nom de chaque opération d'un raffinement doit correspondre au nom d'une opération de l'abstraction correspondante. Cette opération peut prendre la forme d'une opération propre ou d'une opération promue, indépendamment du choix qui a été fait lors des abstractions du raffinement. Ainsi, chaque opération peut être raffinée par :

- une opération propre dans la clause OPERATIONS, dont le nom est celui de l'opération déclarée dans l'abstraction. Si le nom de l'opération de l'abstraction comporte un préfixe alors le nom de l'opération du raffinement le conserve.
- une opération promue par le raffinement, dont le nom est le nom de l'opération déclarée dans l'abstraction. Si l'opération promue par le raffinement provient d'une instance de machine *incluse* non renommée, alors cette machine doit posséder une opération de même nom. Si l'opération promue par le raffinement provient d'une instance de machine *incluse* renommée, alors le premier préfixe du nom de l'opération doit correspondre au renommage de l'instance de machine *incluse* par le raffinement et le nom de l'opération sans le premier préfixe doit correspondre à une opération de la machine *incluse*.

Exemple

```
MACHINE
    MA
INCLUDES
    b2.MB
PROMOTES
    b2.op1,
    b2.op2
OPERATIONS
    op3 = ... ;
    op4 = ...
...
END
```

```
MACHINE
    MB
OPERATIONS
    op1 = ... ;
    op2 = ...
...
END
```

```
MACHINE
    MC
OPERATIONS
    op1 = ...
...
END
```

```
MACHINE
    MD
OPERATIONS
    op3 = ...
...
END
```

```
REFINEMENT
    MA_r
REFINES
    MA
INCLUDES
    b2.MC,
    MD
PROMOTES
    b2.op1,
    op3
OPERATIONS
    b2.op2 = ... ;
    op4 = ...
...
END
```

Dans un raffinement, les paramètres formels de chaque opération doivent être identiques à ceux de l'opération raffinée. Chaque paramètre formel conserve le même type que celui qui est défini dans l'abstraction correspondant au raffinement.

Corps d'opération

Le corps d'une opération propre est constitué d'une substitution. Seules les substitutions de niveau raffinement sont autorisées. Il n'est pas nécessaire de typer les paramètres formels de l'opération dans le corps de l'opération. En effet, le nom et le type de ces paramètres sont déterminés dans la machine correspondant au raffinement et restent identiques au cours du raffinement. Le mécanisme du raffinement d'opération se caractérise par plusieurs propriétés concernant le corps de l'opération du raffinement :

- le niveau d'indéterminisme de la substitution doit diminuer par rapport à celui de l'abstraction. Il s'agit donc, là où l'abstraction comportait des choix, d'apporter peu à peu des solutions afin de lever l'indéterminisme. Dans le dernier raffinement, l'implantation, l'indéterminisme doit avoir complètement disparu.
- les préconditions peuvent être affaiblies par rapport à l'abstraction. Si une précondition est présente en en-tête d'une opération de raffinement, il faut prouver que cette précondition est plus faible que celle de la machine abstraite. Dans l'implantation, les préconditions doivent avoir disparu.
- la structure de la substitution doit évoluer vers l'utilisation de substitutions de plus en plus concrètes. Les substitutions concrètes sont les substitutions qui peuvent être implémentées par un programme informatique. Dans l'implantation, seules les substitutions concrètes sont acceptées.

Les propriétés décrites ci-dessus permettent de raffiner par étapes une opération jusqu'à l'obtention d'un programme informatique. Pour que le raffinement d'une opération possède un sens en B, il faut démontrer que pour chaque raffinement, le corps de l'opération est cohérent avec ce qui a été spécifié dans l'abstraction.

Utilisation dans une implantation

La clause OPERATIONS d'une implantation suit les mêmes principes que ceux d'un raffinement, mais les substitutions qui composent son corps doivent avoir les caractéristiques suivantes : les substitutions employées doivent être déterministes, les préconditions doivent avoir disparu et les substitutions doivent être concrètes, c'est-à-dire qu'elles doivent pouvoir être exécutées par un programme.

En plus du raffinement de certaines opérations de la machine, la clause OPERATIONS d'une implantation contient le raffinement de toutes les opérations locales spécifiées dans la clause LOCAL_OPERATIONS de l'implantation (cf. §7.24 *La clause LOCAL_OPERATIONS*).

Exemple

```
MACHINE
    MA
OPERATIONS
    res_min, res_max, egal ⟵ Comparer (x1, x2) =
    PRE
        x1 ∈ INT  ∧
        x2 ∈ INT
    THEN
        res_min := min ({x1, x2})  ||
        res_max := max ({x1, x2})  ||
        egal := bool (x1 = x2)
    END
END
```

```
REFINEMENT
    MA_r
REFINES
    MA
OPERATIONS
    res_min, res_max, egal ⟵ Comparer (x1, x2) =
    BEGIN
        IF  x1 ≤ x2 THEN
                res_min, res_max :=  x1, x2
        ELSE
                res_min, res_max :=  x2, x1
        END  ;
        egal := bool (x1 = x2)
    END
END
```

7.24 La clause LOCAL_OPERATIONS

Syntaxe

Clause_operations_locales	::=	"LOCAL_OPERATIONS" *Opération*$^{+";"}$
Opération	::=	*Entête_opération* "=" *Substitution_corps_opération*
Entête_opération	::=	[Ident$^{+","}$ "⟵"] *Ident_ren* ["(" Ident$^{+","}$ ")"]

Substitution_corps_opération ::=
Substitution_bloc
| *Substitution_identité*
| *Substitution_devient_égal*
| *Substitution_précondition*
| *Substitution_assertion*
| *Substitution_choix_borné*
| *Substitution_conditionnelle*
| *Substitution_sélection*
| *Substitution_cas*
| *Substitution_any*
| *Substitution_let*
| *Substitution_devient_elt_de*
| *Substitution_devient_tel_que*
| *Substitution_variable_locale*
| *Substitution_appel_opération*

Restrictions

1. Les paramètres formels d'une opération locale doivent être deux à deux distincts.
2. Les opérations locales ne doivent pas être renommées.
3. Les paramètres d'entrée d'une opération locale doivent être typés, dans le prédicat de la substitution précondition qui doit débuter le corps de l'opération locale, par des prédicats de typage (cf. §3.7 *Typage des paramètres d'entrée d'opération*) situés au plus haut niveau d'analyse syntaxique dans une série de conjonctions. Ces paramètres d'entrée ne peuvent pas être utilisés dans le prédicat de la substitution précondition avant d'avoir été typés.
4. Les paramètres de sortie d'une opération locale doivent être typés dans le corps de l'opération par des substitutions de typage (cf. §3.9 *Typage des variables locales et des paramètres de sortie d'opération*). Ces paramètres de sortie ne peuvent pas être utilisés dans le corps de l'opération avant d'avoir été typés.

Description

Les opérations locales d'une implantation sont dites locales car elles ne sont utilisables que par les opérations (non locales ou locales) de cette implantation, mais pas par des composants extérieurs à l'implantation. Une opération locale est spécifiée dans la clause LOCAL_OPERATIONS et implantée dans la clause OPERATIONS, avec l'implantation des opérations non locales et non promues.

Les opérations locales partagent de nombreuses caractéristiques avec les opérations non locales (cf. §7.23 *La clause OPERATIONS*) : elles peuvent faire évoluer des variables à l'aide de substitutions ; elles peuvent également posséder des paramètres d'entrée et de sortie. Elles diffèrent des opérations non locales par leur raffinement : elles sont spécifiées et implantées dans une même implantation, et par leur visibilité : elles ne sont accessibles (sous la forme d'appels d'opération, cf. §6.16 *Substitution appel* *d'opération*) que par les opérations de l'implantation dans laquelle elles sont définies.

Il faut démontrer que les spécifications d'opérations locales préservent l'invariant des machines *importées* (note : l'invariant de l'implantation n'est pas forcément vrai lors de l'appel d'une opération locale depuis l'intérieur d'une opération de l'implantation). Il faut également démontrer que l'implantation de chaque opération locale (cf. §7.23 *La clause OPERATIONS*) est cohérente avec la spécification de l'opération locale.

Utilisation

Les opérations locales servent à factoriser l'écriture d'un projet B. Une opération locale se définit dans une implantation par sa spécification et son implantation. Comme toujours dans la Méthode B, les appels à une opération locale seront remplacés par la spécification de l'opération locale lors de la preuve et par un appel à leur implémentation dans le programme informatique associé au projet.

La spécification d'une opération locale nécessite des substitutions de machine abstraite, comme la spécification d'une opération non locale. En particulier, la substitution simultanée est autorisée, mais pas la substitution séquencement. Les constantes et variables abstraites du raffinement de l'implantation et des instances de machines *vues* ou *importées* par l'implantation sont accessibles dans la spécification de l'opération. De plus, les variables *importées* sont modifiables directement par la spécification de l'opération locale.

L'implantation d'une opération locale est située dans la clause OPERATIONS, avec l'implantation des opérations de la machine non promues par l'implantation. Elles doivent respecter les mêmes règles que ces opérations non locales. En particulier, la substitution simultanée est interdite, la substitution séquencement est autorisée et les constantes et variables abstraites ne sont pas accessibles dans les instructions.

Une opération locale peut être appelée par les implémentations des opérations non locales de l'implantation. Elle ne peut pas être appelée par l'initialisation de l'implémentation. Elle possède des droits similaires à ceux des opérations non locales de l'implantation. Elle peut notamment modifier directement les variables concrètes de l'implantation. Elle peut également modifier les variables des instances de machines *importées* (directement dans la spécification des opérations locales et indirectement, par des appels d'opération dans l'implémentation des opérations locales, cf. *Modèle équivalent* ci-après). Si une opération locale est appelée plusieurs fois, on factorise un traitement commun.

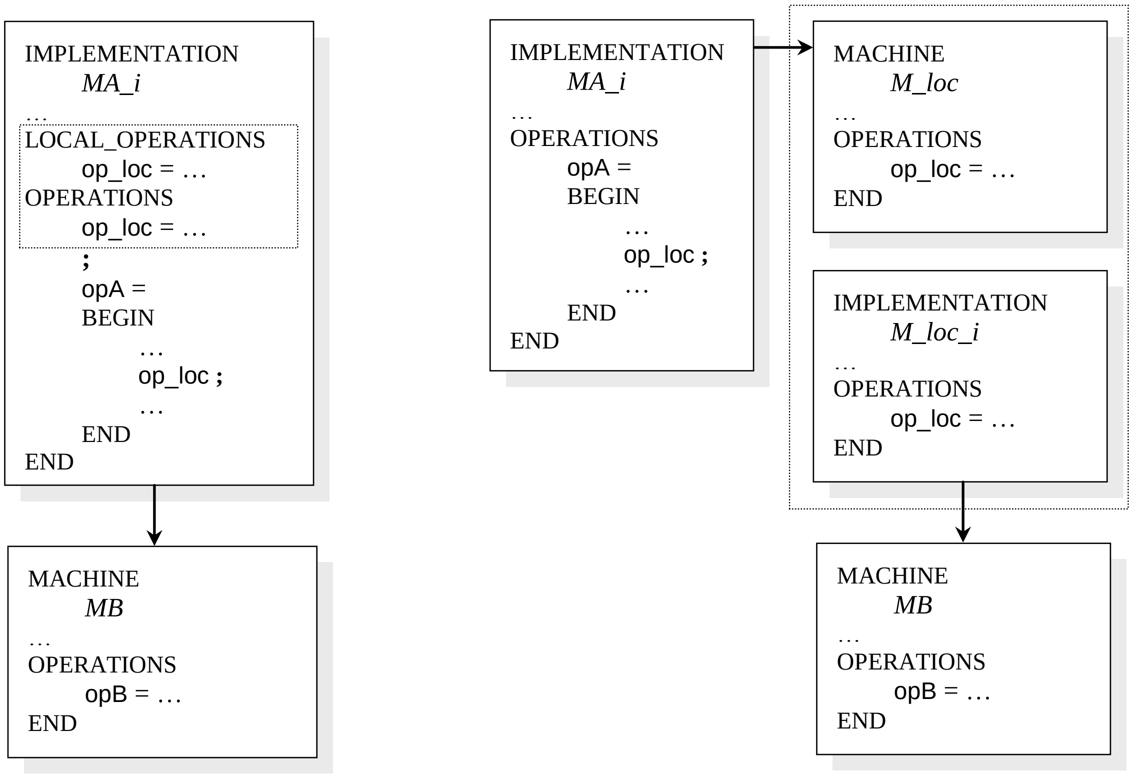
Exemple

```
IMPLEMENTATION
      MA_i
…
LOCAL_OPERATIONS
      max_y =
      BEGIN
            x0  :=  max (y1, y2)
      END
OPERATIONS
      max_y =
      IF  y1 ≥ y2  THEN
            x0 := y1
      ELSE
            x0 := y2
      END
      ;
      OpA =
      BEGIN
            …
            max_y ;
            …
      END
END
```

Modèle équivalent

Le sens des opérations locales est décrit par le modèle équivalent suivant. Soit *MA_i* une implantation qui définit l'opération locale *op_loc* et qui *importe* la machine *MB*. Le principe général du modèle est équivalent est le suivant : l'implantation *MA_i' importe* une machine *M_loc* contenant la spécification de *op_loc* et la machine *M_loc* est raffinée par l'implantation *M_loc_i* qui contient l'implantation de *op_loc* et qui fait un *extends* de *MB*. C'est ce qui est représenté par le schéma suivant. En entrant plus dans les détails, on obtient *MA_i'* à partir de *MA_i* en supprimant les déclarations de variables concrètes et l'initialisation. On recopie tous les éléments constituant *MB'* dans la machine *M_loc*. Dans le cas où *MB* inclut une machine *MC*, alors on recopie également dans *M_loc* toutes les données de *MC* et on effectue l'expansion des appels aux opérations de *MC*. On recopie également dans *M_loc* toutes les variables concrètes présentes dans *MA_i*. L'invariant de *M_loc* se compose de l'invariant de *MB*, de celui de *MC* et du typage B des variables concrètes de *MA_i*. L'initialisation de *M_loc* comprend l'initialisation de *MC*, puis l'initialisation de *MB*, puis l'initialisation de *MA_i*. Enfin, les instructions du corps des opérations de *MA_i* qui ne sont pas des appels d'opérations sont transformées en opération dans *M_loc* et dans *M_loc_i*.

Les nouvelles obligations de preuve obtenues pour les opérations locales découlent directement des obligations de preuve classiques des opérations du module d'opérations locales.



La factorisation obtenue est claire : on gagne l'écriture du module d'opérations locales.

7.25 Spécificités du B0

On appelle B0, la partie du langage B permettant de décrire les opérations et les données des implantations. Le B0 équivaut à un langage de programmation informatique, manipulant des données concrètes, alors que le langage B est un langage de spécification et de programmation.

Les données concrètes présentes dans le B0 sont les constantes concrètes, les variables concrètes, les paramètres d'entrée et de sortie d'opération, les paramètres de machines, les variables locales, les ensembles abstraits et les ensembles énumérés ainsi que leurs éléments. La nature de ces données est décrite au §3.4 *Types et contraintes des données concrètes*.

Afin de bien marquer la différence entre les langages B et B0, on adopte un nouveau vocabulaire pour désigner les productions de la grammaire du B0. Les substitutions concrètes sont appelées instructions (cf. §7.25.4). Les prédicats concrets sont appelés conditions (cf. §7.25.3) et les expressions concrètes sont appelées termes (cf. §7.25.2).

7.25.1 Contrôle des tableaux en B0

Description

Pour garantir que des tableaux concrets (cf. §3.4 *Types et contraintes des données concrètes*) soient traduisibles, on ajoute aux contrôles de type concernant les prédicats, les expressions et les substitutions destinés à être traduits, un contrôle de compatibilité B0, comme défini ci-dessous.

Restriction

1. Deux tableaux concrets sont compatibles en B0 s'ils ont le même type et s'ils ont reçu syntaxiquement le même domaine de définition lors de leur typage. Le domaine de définition d'un tableau est déterminé soit directement lorsque le tableau est typé dans un prédicat de typage qui définit explicitement ce domaine, soit par inférence si le tableau est typé à l'aide d'un autre tableau.

Utilisation

Deux tableaux concrets peuvent ne pas être compatibles pour un programme informatique alors qu'ils sont de même type. Ceci se produit lorsque certains ensembles indices des tableaux sont des intervalles de valeurs différentes.

Par exemple, les tableaux concrets $Tab1 \in (1 \,..\, 5) \rightarrow \mathsf{INT}$ et $Tab2 \in (1 \,..\, 10) \rightarrow \mathsf{INT}$ sont de même type ($\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$), mais ils ne peuvent pas servir de valeur à une même variable informatique, puisqu'ils sont de taille différente.

D'après la restriction énoncée ci-dessus, les tableaux concrets $Tab1$ et $Tab2$ ne sont pas compatibles en B0 puisque leurs domaines de définition $(1 \,..\, 5)$ et $(1 \,..\, 10)$ sont syntaxiquement distincts.

Ce contrôle est une condition suffisante mais pas nécessaire afin d'assurer que les valeurs de deux tableaux concrets soient compatibles. En effet, si $c1$ et $c2$ sont deux constantes concrètes désignant des entiers positifs égaux, alors les tableaux $Tab3 \in (0 \,..\, c1) \rightarrow \mathsf{INT}$ et $Tab4 \in (0 \,..\, c2) \rightarrow \mathsf{INT}$ ne seront pas compatibles en B0, même si $c1$ et $c2$ sont égales.

7.25.2 Les termes

Syntaxe

Terme ::= *Terme_simple*
| *Expression_arithmétique*
| *Terme_record*
| *Terme_record* ("’" Ident)$^{+}$

Terme_simple ::=
Ident_ren
| *Entier_lit*
| *Booléen_lit*
| *Ident_ren* ("’" Ident)$^{+}$

Entier_lit ::= Entier_littéral
| "MAXINT"
| "MININT"

Booléen_lit ::= "FALSE"
| "TRUE"

Expression_arithmétique ::=
Entier_lit
| *Ident_ren*
| *Ident_ren* "(" *Terme*$^{+","}$ ")"
| *Ident_ren* ("’" Ident)$^{+}$
| *Expression_arithmétique* "+" *Expression_arithmétique*
| *Expression_arithmétique* "-" *Expression_arithmétique*
| "-" *Expression_arithmétique*
| *Expression_arithmétique* "×" *Expression_arithmétique*
| *Expression_arithmétique* "/" *Expression_arithmétique*
| *Expression_arithmétique* "mod" *Expression_arithmétique*
| *Expression_arithmétique*$^{\textit{Expression\_arithmétique}}$
| "succ" "(" *Expression_arithmétique* ")"
| "pred" "(" *Expression_arithmétique* ")"
| "(" *Expression_arithmétique* ")"

Terme_record ::=
"rec" "(" ([Ident ":"] (*Terme* | *Expr_tableau*))$^{+","}$ ")"

Expr_tableau ::=
Ident
| "{" (*Terme_simple*$^{+"\mapsto"}$ "↦" *Terme*)$^{+","}$ "}"
| *Ensemble_simple*$^{+"\times"}$ "×" "{" *Terme* "}"

Intervalle_B0 ::=
Expression_arithmétique ".." *Expression_arithmétique*
| *Ensemble_nombres_B0*

Ensemble_nombres_B0 ::=
"NAT"
| "NAT_1"
| "INT"

Description

Les termes représentent la restriction des expressions du langage B utilisables en B0. Les termes peuvent être implémentés par un programme informatique. Ils sont utilisés au sein des instructions et des conditions.

Les termes doivent être du type des variables concrètes (cf. §3.6 *Typage des variables concrètes*).

L'utilisation directe, dans les instructions, de termes nécessite de prouver que les termes sont bien définis et peuvent être correctement implémentés dans un langage de programmation classique. Pour cela, les obligations de preuves suivantes devront être démontrées :

- lors de l'utilisation, dans une expression B0, d'une donnée de type entier, il faut prouver que la donnée appartient à INT (défini par MININT .. MAXINT) qui est l'ensemble des entiers concrets. En effet, on fait l'hypothèse que sur la machine cible sur laquelle s'exécute le projet, il est possible de représenter directement tout entier compris entre MININT et MAXINT sans qu'il se produise un débordement.
- lors de l'utilisation dans une expression B0 d'un opérateur arithmétique, il faut prouver que ses opérandes appartiennent au domaine de définition de l'opérateur en B0 et que le résultat appartient à INT. Les opérateurs arithmétiques utilisables dans les termes ainsi que leurs domaines de définition sont donnés dans le tableau ci-dessous :

Expression B0 arithmétique		Condition		
Addition en B0	$a + b$	$a \in$ INT	$\wedge\ b \in$ INT	$\wedge\ a + b \in$ INT
Soustraction en B0	$a - b$	$a \in$ INT	$\wedge\ b \in$ INT	$\wedge\ a - b \in$ INT
Moins unaire en B0	$- a$	$a \in$ INT		$\wedge\ - a \in$ INT
Multiplication en B0	$a \times b$	$a \in$ INT	$\wedge\ b \in$ INT	$\wedge\ a \times b \in$ INT
Division entière en B0	a / b	$a \in$ INT	$\wedge\ b \in$ INT - {0}	$\wedge\ a / b \in$ INT
Modulo en B0	a mod b	$a \in$ NAT	$\wedge\ b \in$ NAT_1	$\wedge\ a$ mod $b \in$ INT
Puissance en B0	a^b	$a \in$ INT	$\wedge\ b \in$ NAT	$\wedge\ a^b \in$ INT
Successeur en B0	succ (a)	$a \in$ INT		$\wedge$ succ (a) $\in$ INT
Prédécesseur en B0	pred (a)	$a \in$ INT		$\wedge$ pred (a) $\in$ INT

- lors de l'accès dans une instruction à un élément d'un tableau concret (on rappelle qu'en B un tableau est une fonction totale), il faut prouver que l'indice utilisé appartient au domaine de définition du tableau.

7.25.3 Les conditions

Syntaxe

Condition ::= *Terme_simple* "=" *Terme_simple*
| *Terme_simple* "≠" *Terme_simple*
| *Terme_simple* "<" *Terme_simple*
| *Terme_simple* ">" *Terme_simple*
| *Terme_simple* "≤" *Terme_simple*
| *Terme_simple* "≥" *Terme_simple*
| *Condition* "∧" *Condition*
| *Condition* "∨" *Condition*
| "¬" "(" *Condition* ")"
| "(" *Condition* ")"

Description

Les conditions représentent la restriction des prédicats du langage B utilisables en B0. Les conditions peuvent être évaluées par un langage informatique. Elles sont utilisées en B0 comme condition de branchement des instructions conditionnelles IF et comme condition d'arrêt des instructions de boucle « tant que ».

Dans le cas des prédicats d'égalité et d'inégalité portant sur des tableaux, on rappelle que les tableaux doivent, bien sûr, avoir le même type, mais qu'ils doivent aussi avoir le même domaine de définition (cf. §7.25.1 *Contrôle des tableaux en B0*).

7.25.4 Les instructions

Syntaxe

Instruction ::=
Instruction_bloc
| *Instruction_variable_locale*
| *Substitution_identité*
| *Instruction_devient_égal*
| *Instruction_appel_opération*
| *Instruction_conditionnelle*
| *Instruction_cas*
| *Instruction_assertion*
| *Instruction_séquence*
| *Substitution_tant_que*

Instruction_bloc ::=
"BEGIN" *Instruction* "END"

Instruction_variable_locale ::=
"VAR" Ident$^{+","}$ "IN" *Instruction* "END"

Instruction_devient_égal ::=
Ident_ren ["(" *Terme*$^{+","}$ ")"] ":=" *Terme*
| *Ident_ren* ":=" *Expr_tableau*
| *Ident_ren* ("'" Ident)$^{+}$ ":=" *Terme*

Instruction_appel_opération ::=
[*Ident_ren*$^{+","}$ "←"] *Ident_ren* ["(" (*Terme* | *Chaîne_lit*)$^{+","}$ ")"]

Instruction_séquence ::=
Instruction ";" *Instruction*

Instruction_conditionnelle ::=
"IF" *Condition* "THEN" *Instruction*
("ELSIF" *Condition* "THEN" *Instruction*)*
["ELSE" *Instruction*]
"END"

Instruction_cas ::=
"CASE" *Terme_simple* "OF"
"EITHER" *Terme_simple*$^{+","}$ "THEN" *Instruction*
("OR" *Terme_simple*$^{+","}$ "THEN" *Instruction*)*
["ELSE" *Instruction*]
"END"
"END"

Instruction_assertion ::=
"ASSERT" *Prédicat* "THEN" *Instruction* "END"

Substitution_tant_que ::=
"WHILE" *Condition* "DO" *Instruction*
"INVARIANT" *Prédicat*
"VARIANT" *Expression*
"END"

Description

Les instructions représentent une restriction des substitutions du langage B qui peuvent être implémentées par un programme informatique. Les instructions sont utilisées dans le corps de l'initialisation et dans le corps des opérations. Voici les particularités des instructions :

Instruction « devient égal »

Dans une instruction « devient égal », il est seulement permis de réaliser les affectations suivantes :

- affectation d'une donnée scalaire,
- affectation d'une donnée tableau, tous les éléments du tableau doivent recevoir une valeur. La valeur affectée peut être soit une donnée tableau soit un tableau littéral (cf. §7.17 *La clause VALUES*). Les tableaux doivent bien sûr avoir le même type, mais ils doivent aussi avoir le même domaine de définition (cf. §7.25.1 *Contrôle des tableaux en B0*).
- affectation d'un élément de tableau, les indices utilisés pour désigner un élément de tableau doivent être des termes.
- affectation d'un champ, ou d'un sous-champ, d'une donnée record.

Instruction d'appel d'opération

Dans une instruction d'appel d'opération, les paramètres effectifs d'entrée peuvent être soit des termes, soit des chaînes de caractères littérales. Dans le cas où un paramètre effectif d'entrée ou de sortie d'un appel opération est un tableau, le paramètre formel et le paramètre effectif doivent bien sûr avoir le même type, mais ils doivent aussi avoir le même domaine de définition (cf. §7.25.1 *Contrôle des tableaux en B0*).

Instruction CASE

Dans une instruction CASE, l'expression de sélection doit être un terme simple.

Instruction ASSERT

Dans une instruction ASSERT, l'assertion introduite reste un prédicat car elle ne sert pas à la production de code, mais à la preuve de l'implantation.

7.26 Règles d'anticollision d'identificateurs

Les règles d'anticollision d'identificateurs servent à éviter que dans une clause d'un composant, il soit possible d'accéder à plusieurs constituants portant le même nom mais désignant des constituants différents sans savoir lequel est effectivement utilisé.

Les règles d'anticollision dépendent principalement des règles de visibilité entre composants. En effet si un composant M_A *voit* une instance de machine M_B, alors une donnée de M_B accessible par M_A ne doit pas porter le même nom qu'une donnée de M_A.

Les données déclarées dans un prédicat, une substitution ou dans un en-tête d'opération ne participent pas au contrôle d'anticollision. En effet, elles ont une portée limitée respectivement au prédicat, à la substitution et au corps de l'opération dans lequel elles sont déclarées. Si elles portent le même nom qu'un constituant accessible, alors elles le masquent localement.

Machine abstraite

Soit une machine abstraite *Mch*.

La liste *LMch* comprend les identificateurs suivants de *Mch* :

- nom de *Mch*,
- nom des paramètres de *Mch*,
- nom des ensembles abstraits et des ensembles énumérés de *Mch*,
- nom des éléments énumérés et des constantes de *Mch*,
- nom des variables de *Mch*,
- nom des opérations propres de *Mch*.

La liste *LSees* comprend les identificateurs suivants pour chaque machine *vue MSees* par *Mch* :

- nom de *MSees* avec le préfixe de renommage,
- nom des ensembles abstraits et des ensembles énumérés de *MSees*, sans le préfixe de renommage, si plusieurs instances de machines sont *vues*, les noms des données ne doivent pas être répétés,
- nom des éléments énumérés et des constantes de *MSees*, sans le préfixe de renommage, si plusieurs instances de machines sont *vues*, les noms des données ne doivent pas être répétés,
- nom des variables de *MSees* avec le préfixe de renommage,
- nom des opérations de *MSees* avec le préfixe de renommage.

La liste *LInc* comprend les identificateurs suivants pour chaque machine *incluse MInc* par *Mch* :

- nom de *MInc*, avec le préfixe de renommage,
- nom des ensembles abstraits et des ensembles énumérés de *MInc*, sans le préfixe de renommage, si plusieurs instances de machines sont *incluses*, les noms des données ne doivent pas être répétés,
- nom des éléments énumérés et des constantes de *MInc*, sans le préfixe de renommage, si plusieurs instances de machines sont *incluses*, les noms des données ne doivent pas être répétés,

- nom des variables de *MInc*, avec le préfixe de renommage,
- nom des opérations de *MInc*, avec le préfixe de renommage.

La liste *LUses* comprend les identificateurs suivants pour chaque machine *utilisée MUses* par *Mch* :

- nom de *MUses*, avec le préfixe de renommage,
- nom des paramètres de *MUses*, avec le préfixe de renommage,
- nom des ensembles abstraits et des ensembles énumérés de *MUses*, sans le préfixe de renommage, si plusieurs instances de machines sont *utilisées*, les noms des données ne doivent pas être répétés,
- nom des éléments énumérés et des constantes de *MUses*, sans le préfixe de renommage, si plusieurs instances de machines sont *utilisées*, les noms des données ne doivent pas être répétés,
- nom des variables de *MUses*, avec le préfixe de renommage.

Règle d'anticollision

Les noms de la liste $LMch \cup LSees \cup LInc \cup LUses$ doivent être deux à deux distincts.

Raffinement

Soit un raffinement *Raf* dont la machine abstraite est *Mch*.

La liste *LRaf* comprend les identificateurs suivants de *Raf* :

- nom de *Mch*,
- nom des paramètres de *Raf*,
- nom des ensembles abstraits et des ensembles énumérés de *Raf*, sauf ceux qui sont homonymes à un élément de même nature d'une instance de machine *incluse* ou *vue*,
- nom des éléments énumérés et des constantes de *Raf*, sauf ceux qui sont homonymes à un élément de même nature d'une instance de machine *incluse* ou *vue*,
- nom des variables de *Raf*, sauf les variables concrètes provenant de l'abstraction de *Raf* et homonymes à des variables concrètes d'une instance de machine *incluse*,
- nom des opérations propres de *Raf*,
- nom des constantes abstraites de l'abstraction de *Raf* disparaissant dans *Raf*, sauf celles qui sont homonymes à des constantes abstraites d'une instance de machine *incluse* ou *vue*,
- nom des variables abstraites de l'abstraction de *Raf* disparaissant dans *Raf*, sauf celles qui sont homonymes à des variables abstraites d'une instance de machine *incluse*.

La liste *LSees* comprend les identificateurs suivants pour chaque machine *vue MSees* par *Raf* :

- nom de *MSees*, avec le préfixe de renommage,
- nom des ensembles abstraits et des ensembles énumérés de *MSees*, sans le

préfixe de renommage, si plusieurs instances de machines sont *vues*, les noms des données ne doivent pas être répétés,

- nom des éléments énumérés et des constantes de *MSees*, sans le préfixe de renommage, si plusieurs instances de machines sont *vues*, les noms des données ne doivent pas être répétés,
- nom des variables de *MSees*, avec le préfixe de renommage,
- nom des opérations de *MSees*, avec le préfixe de renommage.

La liste *LInc* comprend les identificateurs suivants pour chaque machine *incluse MInc* par *Raf* :

- nom de *MInc*, avec le préfixe de renommage,
- nom des ensembles abstraits et des ensembles énumérés de *MInc*, sans le préfixe de renommage, si plusieurs instances de machines sont *incluses*, les noms des données ne doivent pas être répétés,
- nom des éléments énumérés et des constantes de *MInc*, sans le préfixe de renommage, si plusieurs instances de machines sont *incluses*, les noms des données ne doivent pas être répétés,
- nom des variables de *MInc*, avec le préfixe de renommage,
- nom des opérations de *MInc*, avec le préfixe de renommage.

Règle d'anticollision

Les noms de la liste $LRaf \cup LSees \cup LInc$ doivent être deux à deux distincts.

Implantation

Soit une implantation *Imp* dont la machine abstraite est *Mch*.

La liste *LImp* comprend les identificateurs suivants de *Imp* :

- nom de *Mch*,
- nom des paramètres de *Imp*,
- nom des ensembles abstraits, sauf ceux qui proviennent de l'abstraction de *Imp* et qui sont homonymes à des ensembles abstraits d'une instance de machine *importée* ou *vue*,
- nom des ensembles énumérés de *Imp* et de leurs éléments énumérés, sauf pour les ensembles énumérés qui proviennent de l'abstraction de *Imp* et qui sont homonymes à un ensemble énuméré d'une machine *importée* ou *vue*,
- nom des constantes concrètes de *Imp*, sauf celles qui proviennent de l'abstraction de *Imp* et qui sont homonymes à des constantes concrètes d'une instance de machine *importée* ou *vue*,
- nom des variables concrètes de *Imp*, sauf celles qui proviennent de l'abstraction de *Imp* et qui sont homonymes à des variables concrètes d'une instance de machine *importée*,
- nom des opérations de la clause OPERATIONS de *Imp* (il s'agit des opérations non promues et non locales ainsi que des opérations locales),
- nom des constantes abstraites de l'abstraction de *Imp*, sauf celles qui sont homonymes à des constantes abstraites d'une instance de machine *importée*.

La liste *LImports* comprend les identificateurs suivants pour chaque machine *importée MImports* par *Imp* :

- nom de *MImports*, avec le préfixe de renommage,
- nom des ensembles abstraits et des ensembles énumérés de *MImports*, sans le préfixe de renommage, si plusieurs instances de machines sont *importées*, les noms des données ne doivent pas être répétés,
- nom des éléments énumérés et des constantes de *MImports*, sans le préfixe de renommage, si plusieurs instances de machines sont *importées*, les noms des données ne doivent pas être répétés,
- nom des variables de *MImports*, avec le préfixe de renommage,
- nom des opérations de *MImports*, avec le préfixe de renommage.

La liste *LSees* comprend les identificateurs suivants pour chaque machine *vue MSees* par *Imp* :

- nom de *MSees*, avec le préfixe de renommage,
- nom des ensembles abstraits et des ensembles énumérés de *MSees*, sans le préfixe de renommage, si plusieurs instances de machines sont *vues*, les noms des données ne doivent pas être répétés,
- nom des éléments énumérés et des constantes de *MSees*, sans le préfixe de renommage, si plusieurs instances de machines sont *vues*, les noms des données ne doivent pas être répétés,
- nom des variables de *MSees*, avec le préfixe de renommage,
- nom des opérations de *MSees*, avec le préfixe de renommage.

Règle d'anticollision

Les noms de la liste $LImp \cup LImports \cup LSees$ doivent être deux à deux distincts.

8 ARCHITECTURE B

8.1 Introduction

Un développement complet en B se déroule dans le cadre d'un projet B. Un projet permet de modéliser de manière formelle un système de nature quelconque. La finalité du projet B est de produire un programme exécutable. La sûreté de fonctionnement de cet exécutable est étudiée en détail par la méthode B. La construction d'un projet B se fait à l'aide du développement de modules B.

8.2 Module B

Présentation

Un module B permet de modéliser un sous-système ; il constitue une partie d'un projet B. Les modules sont constitués par des composants B. Les trois sortes de composants B existant sont la machine abstraite, le raffinement et l'implantation. Un module possède les propriétés suivantes : il comprend toujours une machine abstraite, qui représente la spécification du module. Il peut posséder une implantation et éventuellement des raffinements. Enfin, il peut posséder un code associé. Il existe trois sortes de modules qui se définissent en fonction de leurs propriétés. Il s'agit des modules développés par raffinements successifs d'une machine abstraite, des modules de base et des modules abstraits. Ces modules sont décrits dans le tableau ci-dessous.

Module **Propriétés**	**Module développé**	**Module de base**	**Module abstrait**
Possède une machine abstraite	oui	oui	oui
Possède une implantation et éventuellement des raffinements	oui	non	non
Possède un code associé	oui (par traduction)	oui (manuellement)	non

Machine abstraite

Une machine abstraite contient la description de la spécification d'un module B. À ce titre, le langage B constitue donc un langage de spécification à part entière. Seule la machine abstraite d'un module est accessible par les modules externes. Par abus de langage, on emploie parfois le terme machine abstraite ou plus simplement machine à la place de module. En effet, d'une part le nom du module et de sa machine abstraite sont confondus et d'autre part l'interface du module, c'est-à-dire la partie accessible de l'extérieur, est commune au module et à sa machine abstraite.

Une machine abstraite comprend des liens (cf. §8.3 *Liens entre composants*), une partie statique et une partie dynamique. La partie statique est formée de données prenant la forme d'ensembles, de constantes, de variables ou de paramètres et par les propriétés de ces données. Une donnée est un objet mathématique faisant partie de la boite à outils mathématique du langage B (cf. chapitre 5 *Expressions*), comme par exemple un scalaire, un ensemble, une fonction ou une suite. Les données sont encapsulées dans la machine abstraite. La partie dynamique permet de manipuler les données. Elle est constituée de l'initialisation qui permet de donner une valeur initiale aux variables et d'opérations qui correspondent à des services offerts par la machine pour manipuler les variables. On nomme invariant les propriétés des variables de la machine. L'invariant doit être établi lors de l'initialisation de la machine et il doit être préservé lors de l'appel d'une opération de la machine. L'invariant constitue donc l'énoncé des propriétés de sécurité de la machine.

Raffinement

Le raffinement d'une machine abstraite est un composant qui conserve la même interface et le même comportement que la machine abstraite mais qui reformule les données et les opérations de la machine à l'aide de données plus concrètes. Le raffinement permet également d'enrichir ce qui a été spécifié dans la machine abstraite. Lors du raffinement, les ensembles et les données concrètes d'une machine sont conservés. Les données raffinables sont raffinées, ce qui signifie qu'elles peuvent être conservées, disparaître ou changer de forme. De nouvelles données peuvent être introduites. Le corps des opérations doit également être raffiné : chaque opération raffinée doit réaliser ce qui est spécifié dans l'abstraction, à l'aide des données du raffinement et de substitutions plus concrètes et plus déterministes.

Un premier raffinement peut à son tour être raffiné par un autre raffinement selon les principes indiqués ci-dessus. Plusieurs niveaux de raffinement peuvent ainsi être utilisés afin de reformuler progressivement, par étapes successives, la machine abstraite.

Implantation

Une implantation est un composant B qui constitue le niveau ultime de raffinement d'une machine abstraite. Elle utilise largement un sous-ensemble du langage B, appelé B0, semblable à un langage de programmation informatique. Les données d'une implantation doivent être des données concrètes (scalaires, tableaux, chaînes de caractères) implémentables directement dans un langage informatique évolué (cf. §3.4 *Types et contraintes des données concrètes*). Le corps des opérations d'une implantation doit être constitué par des substitutions concrètes, appelées instructions, exécutables directement dans un langage informatique évolué (cf. §7.24 La clause LOCAL_OPERATIONS). Ces propriétés font qu'il est possible de produire systématiquement un programme informatique à partir du B0 d'un projet B sémantiquement correct. Pour obtenir une meilleure intégration dans n'importe quel système informatique, le B0 est traduit automatiquement dans un langage informatique évolué comme Ada ou C++.

Exemple

Le schéma suivant représente graphiquement un module développé complet. *ModuleA* représente à la fois le nom du module et le nom de la machine abstraite représentant la spécification du module. *ModuleA_r1* et *ModuleA_r2* sont les noms des raffinements successifs de *ModuleA*. *ModuleA_i* est le nom de l'implantation de *ModuleA*.

ModuleA
ModuleA_r1
ModuleA_r2
ModuleA_i

Module de base

Un module de base, encore appelé machine de base, désigne un module B qui n'est composé que d'une machine abstraite. Une machine de base correspond à une feuille dans le graphe d'importation d'un projet. Contrairement aux autres modules qui sont raffinés et peuvent être traduits, il n'est pas traduit mais doit posséder un code associé qui implémente directement ses données et ses services. En effet, comme la machine abstraite peut comporter des données abstraites et des substitutions abstraites, éventuellement non déterministes, il n'est pas possible de produire de manière systématique un programme à partir de ces seules spécifications.

Les machines de base peuvent servir d'interface avec un code existant ou bien avec des fonctionnalités de bas niveau qui n'existent pas dans le langage B, comme les fonctions système. Les fonctions d'entrées/sorties constituent un exemple typique de fonctionnalités interfacées à l'aide de machines de base.

Module abstrait

Un module abstrait est composé d'une machine abstraite qui n'est pas raffinée et qui ne possède pas de code associé. La seule utilisation d'un module abstrait dans un projet B consiste à l'*inclure* (cf. *lien INCLUDES*) dans une machine abstraite ou dans un raffinement sans jamais l'*importer* (cf. *lien IMPORTS*) dans le projet. Il constitue donc un intermédiaire de raisonnement.

8.3 Projet B

Présentation

Un projet B désigne un ensemble complet d'instances de modules B. Les composants de ces instances de modules sont reliés par des liens. Les liens doivent respecter certaines règles.

Instanciation et renommage

Une instance de module est la copie d'une machine abstraite. L'instanciation permet de réutiliser plusieurs fois une machine abstraite dans un même projet. Chaque instance de machine abstraite possède un espace de données propre qui contient les valeurs des données modifiables de la machine. Ces données sont propres à l'instance, il s'agit des variables (cf. §7.18 *La clause CONCRETE_VARIABLES* et §7.19 *La clause ABSTRACT_VARIABLES*) et des paramètres de la machine (cf. §7.5 *La clause CONSTRAINTS*). Les constantes d'une machine sont propres à la machine puisque leur valeur est identique dans toutes les instances de machines. Lors de l'appel d'une opération d'une instance de machine, les valeurs des variables et des paramètres de la machine manipulées par l'opération sont celles de l'espace de données de l'instance.

On distingue les instances locales à un composant et les instances globales au projet. Les premières sont créées par les liens INCLUDES (cf. *lien INCLUDES*) en phase de spécification ou de raffinement. Elles constituent des espaces de données locaux au composant car seulement accessibles par celui-ci. Les secondes sont créées par les liens IMPORTS (cf. *lien IMPORTS*) en phase d'implantation et constituent les espaces de données globaux au projet car elles sont accessibles depuis l'ensemble du projet à l'aide du lien SEES (cf. *lien SEES*).

Chaque instance possède un nom qui lui est propre. Ce nom peut être soit le nom de la machine sans renommage, soit un identificateur, appelé préfixe de renommage, suivi d'un point et du nom de la machine. Dans le cas d'une instance sans renommage, l'instance et la machine abstraite ont le même nom, mais elles ne doivent pas être confondues. L'instanciation sans renommage représente le cas le plus fréquent dans un projet B puisque les machines qui ne sont instanciées qu'une seule fois n'ont pas besoin d'être renommées (on peut donc choisir de les instancier sans renommage). Par contre, l'instanciation avec renommage est obligatoire dès qu'une machine est instanciée plusieurs fois, car le nom de chaque instance de machine d'un projet B doit être unique pour identifier les espaces de données.

Si un composant *Cmp* accède à une instance de machine *InstMch*, alors le nom de cette instance influe sur le nom sous lequel seront désignés dans *Cmp* les variables, les opérations et les paramètres de *InstMch* (cf. §7.26 *Règles d'anticollision d'identificateurs*). Si l'instance est sans renommage, alors les variables, les opérations et les paramètres de *InstMch* seront utilisés dans *Cmp* sous le même nom que dans la machine abstraite qui les déclare. Si *InstMch* est renommée alors le nom des variables, des opérations et des paramètres utilisés dans *Cmp* devra être préfixé par le préfixe de renommage de *InstMch* suivi d'un point.

Liens entre composants

Les composants d'un projet B peuvent être reliés par cinq sortes des liens : IMPORTS, SEES, INCLUDES, EXTENDS et USES. Ils sont déclarés dans les clauses de visibilité des composants. En voici une description sommaire :

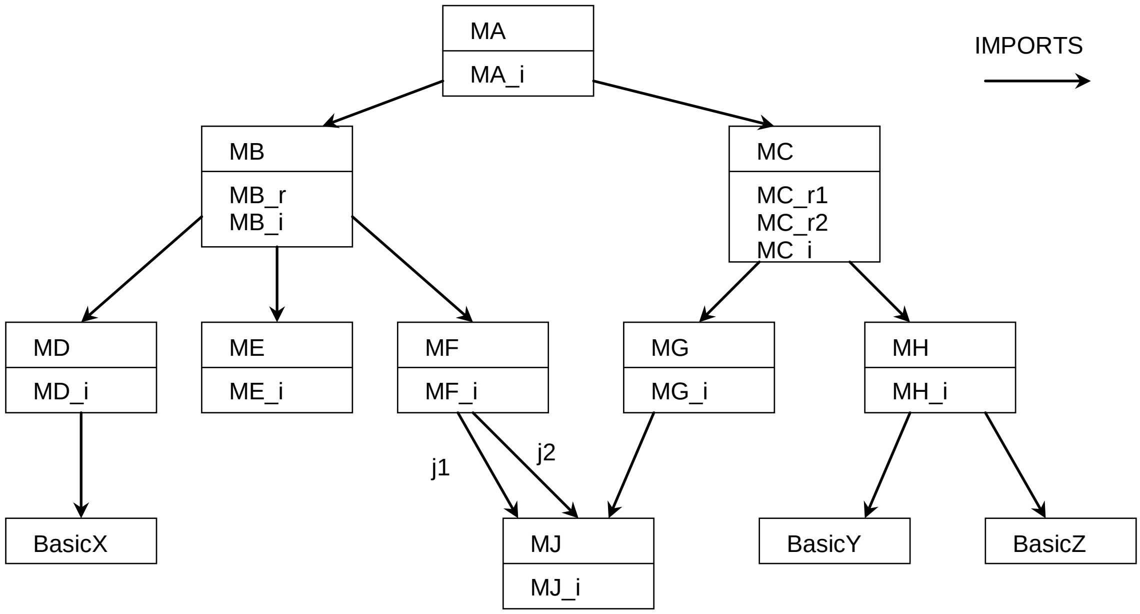
lien IMPORTS

Le lien IMPORTS entre une implantation M_i et une instance de machine M_B permet de créer concrètement l'instance M_B et de disposer entièrement de ses services. On dit que M_i est le père de M_B, car il contrôle entièrement l'écriture des données modifiables de M_B. L'*importation* permet de structurer un projet B en couches, en effet l'implantation d'un module s'implémente par *importation* sur d'autres modules offrant des services de plus bas niveau.

On appelle graphe d'importation d'un projet B, le graphe formé de l'ensemble des modules du projet B et des liens d'*importation* entre les implantations de ces modules. Pour désigner quelle instance de machine est *importée* par un lien, on indique sur le lien le préfixe de renommage de l'instance de machine *importée* et rien s'il n'y a pas de renommage.

Un graphe d'importation doit posséder une unique racine qui joue un rôle particulier. C'est la machine principale du projet. Ses opérations constituent le point d'entrée du projet. À partir de la machine principale, le graphe d'importation s'organise en couches qui représentent les niveaux de décomposition du projet en éléments de plus en plus simples. Une feuille du graphe est soit un module développé terminal, si le module a pu être développé sans recourir aux services de machines de base, soit une machine de base. Le graphe d'importation d'un projet décrit entièrement l'organisation du programme associé au projet puisque chaque module du graphe possède un code associé, qu'il s'agisse de modules développés ou de modules de base.

Le schéma ci-dessous présente un exemple de graphe d'importation d'un projet. Chaque instance de module possède de 0 à *n* fils et un unique père, sauf le module principal qui ne possède pas de père. Les modules *BasicX*, *BasicY* et *BasicZ* représentent des modules de base. Le module *MJ* est *importé* trois fois, une fois sans renommage, une fois avec le renommage *j1* et une fois avec le renommage *j2*.



lien SEES

Le lien SEES est une référence transversale dans le graphe d'importation du projet B qui permet à un composant de *voir* une instance de machine, c'est-à-dire d'accéder en lecture mais pas en écriture aux constituants de l'instance de machine *vue*.

On dit qu'un module dépend d'un autre module, si l'implantation du premier module *voit* ou *importe* une instance du second module. Le graphe de dépendance d'un projet B est le graphe d'importation du projet auquel s'ajoutent les liens SEES. Les liens SEES portent le préfixe de renommage de l'instance de machine *vue*. Ce préfixe peut contenir plusieurs renommages successifs (cf. §7.8 *clause SEES et renommage*).

lien INCLUDES

Le lien INCLUDES entre un composant *MN* (une machine ou un raffinement) et une instance de machine *Minc* permet d'*inclure* dans *MN* les constituants de *Minc* afin de construire un composant plus volumineux. L'*inclusion* crée l'instance de machine *Minc* à un niveau abstrait.

lien EXTENDS

Le lien EXTENDS se comporte comme le lien INCLUDES dans une machine ou un raffinement et comme le lien IMPORTS dans une implantation (cf. §7.11 *La clause EXTENDS*).

lien USES

Lorsqu'un composant *inclut* plusieurs instances de machines, les machines *incluses* peuvent partager les données de l'une d'entre elles, *Mused*, par un lien USES sur *Mused*. La clause USES permet de référencer une instance de machine au sein d'un ensemble d'*inclusion*.

Règles concernant les liens

Les règles concernant les liens entre composants au sein d'un projet sont rassemblées ci-dessous.

Règles sur les liens IMPORTS

1. Une instance de machine ne doit pas être *importée* plus d'une fois dans un projet. Donc pour *importer* plusieurs fois une machine dans un projet, il faut créer plusieurs instances en leur donnant des préfixes de renommage différents.
2. Tout projet complet doit contenir un et un seul module développé qui n'est jamais instancié par *importation* dans le projet. C'est l'unique source du graphe d'*importation* du projet. Ce module s'appelle le module principal du projet.

Règles sur le graphe de dépendance

3. Toute instance de machine *vue* dans un projet doit être *importée* dans le projet.
4. Si une instance de machine est *vue* par un composant d'un module développé, alors les raffinements de ce composant doivent également *voir* cette instance.
5. Un composant d'un module M_A ne peut pas *voir* une instance de module M_B *importée* par une instance de module dont M_A dépend (transitivement). Le schéma ci-dessous illustre les architectures interdites.

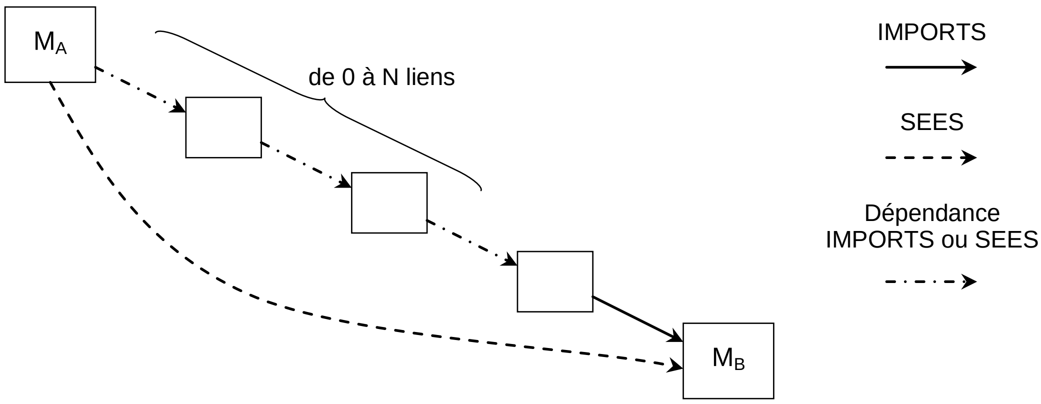


Figure 4 : architecture interdite du lien SEES

6. Un composant ne peut pas posséder plusieurs liens sur une même instance de machine. Par exemple, une implantation ne peut pas *voir* et *importer* une même instance de machine.
7. Il ne doit pas exister de cycle dans le graphe de dépendance d'un projet.
8. Si un composant A *voit* une machine B, il faut pouvoir aller de A vers B en traversant l'arbre d'importation suivant la règle suivante : il faut partir depuis A, ensuite remonter au moins une fois et finalement descendre exactement une fois pour arriver sur B. Autrement dit on ne peut voir que ses frères, oncles, grand-oncles, grand-grand-oncles etc.

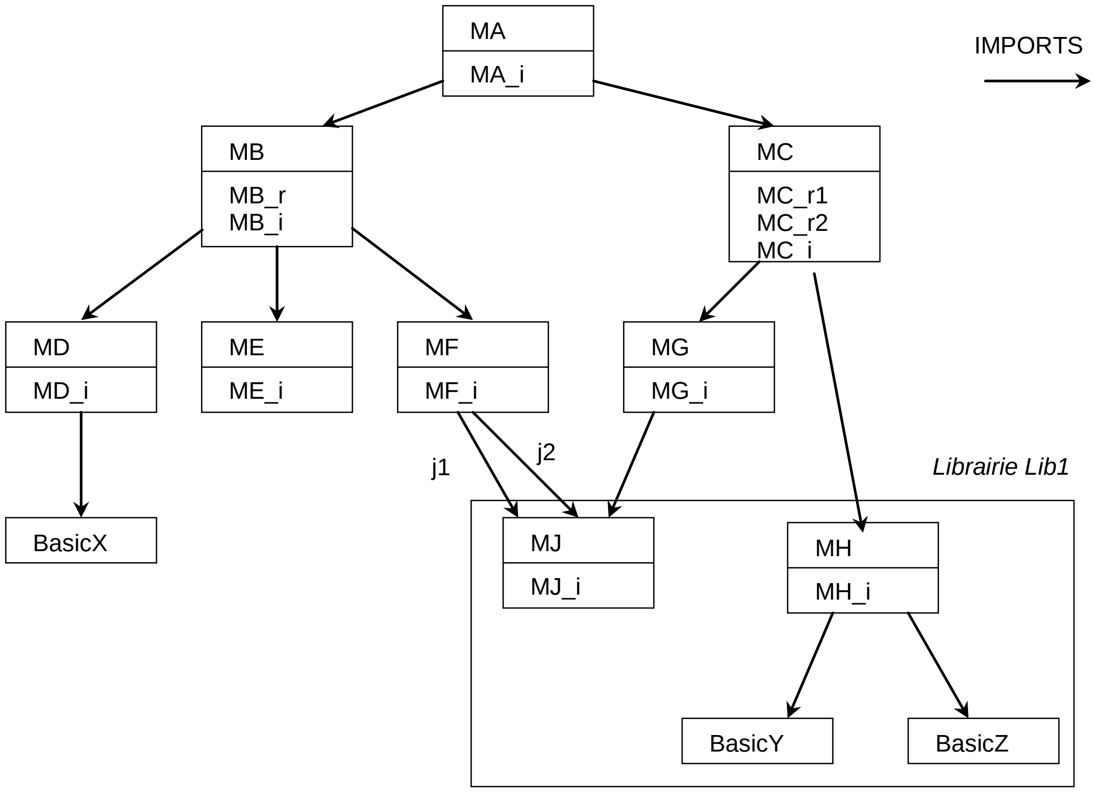
Règles sur les liens USES

9. Si une machine *MA* utilise une instance de machine, *Mused*, alors il doit exister dans le projet une machine qui *inclut* une instance de *MA* et de *Mused*.

8.4 Librairies

Une librairie B est une collection de modules qui peuvent être utilisés dans un projet. La notion de librairie permet de constituer des bibliothèques de modules réutilisables entre plusieurs projets. Elle permet également de décomposer un projet en plusieurs sous-parties, chaque sous-partie étant une librairie. Les modules d'une librairie peuvent eux-mêmes utiliser d'autres librairies.

Un projet B complet peut devenir une librairie. Cependant une librairie ne correspond pas forcément à un projet. En effet, elle peut contenir plusieurs modules principaux. Le schéma ci-dessous donne un exemple de projet utilisant une librairie.



ANNEXES

ANNEXE A MOTS RESERVES ET OPERATEURS

Cette annexe contient la description de l'ensemble des mots réservés et des opérateurs du langage B, triés par ordre ASCII ascendant. L'ordre ASCII est rappelé ci-dessous :

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

Pour chaque mot réservé ou opérateur, sont indiqués :

- sa notation ASCII, éventuellement complétée par son utilisation lorsqu'il existe une correspondance non triviale entre les notations ASCII et mathématique (par exemple, dans le cas de l'opérateur puissance, l'écriture ASCII `x ** y` correspond à la notation mathématique x^y),
- sa notation mathématique, si elle diffère de sa notation ASCII,
- son niveau de priorité. Le niveau de priorité correspond à l'ordre de priorité lors de l'analyse syntaxique. Il suffit de se rappeler que plus un opérateur possède un niveau de priorité élevé, plus il attire ses opérandes. Par exemple, si les opérateurs op_{40} et op_{250} sont respectivement de priorité 40 et 250, alors l'expression $x\ \mathrm{op}_{40}\ y\ \mathrm{op}_{250}\ z$ est analysée comme $x\ \mathrm{op}_{40}\ (y\ \mathrm{op}_{250}\ z)$,
- ses propriétés d'associativité (G pour associatif à gauche ou D pour associatif à droite). Si des opérateurs binaires notés op ont la même priorité, alors : $x\ \mathrm{op}\ y\ \mathrm{op}\ z$ sera analysé comme $(x\ \mathrm{op}\ y)\ \mathrm{op}\ z$ si op est associatif à gauche et comme $x\ \mathrm{op}\ (y\ \mathrm{op}\ z)$ si op est associatif à droite.
- sa description,
- une référence au(x) paragraphe(s) où il est traité.

ASCII	**Math.**	**Pri.**	**As.**	**Description**	**Référence**
!	∀	250		quantificateur universel (quel que soit)	§4.2
"				délimiteur de chaîne de caractères ou de fichier de définition	§5.1, §2.3
#	∃	250		quantificateur existentiel (il existe)	§4.2
$0				valeur précédente d'une donnée	§5.1
%	λ	250		lambda expression	§5.16
&	∧	40	G	conjonction (ET logique)	§4.1
'		250	G	accès à un champ de record	§5.9
(				parenthèse ouvrante	§4.1, §5.1
)				parenthèse fermante	§4.1, §5.1
*	×	190	G	multiplication ou produit cartésien	§3.2, §5.3, §5.7
x ** *y*	x^y	200	D	puissance	§5.3
+		180	G	addition	§5.3
+->	⇸	125	G	fonction partielle	§5.15
+->>	⤀	125	G	surjection partielle	§5.15
,		115	G	virgule	

ASCII	Math.	Pri.	As.	Description	Référence
`-`		180	G	soustraction	§5.3, §5.8
`-`		210		moins unaire	§5.3
`-->`	$\rightarrow$	125	G	fonction totale	§5.15
`-->>`	$\twoheadrightarrow$	125	G	surjection totale	§5.15
`->`	$\rightarrow$	160	G	insertion en tête d'une suite	§5.15
`.`		220	D	renommage ou séparateur de données utilisé dans les opérateurs $\forall$, $\exists$, $\bigcup$, $\bigcap$, Σ, Π, λ	
`..`		170	G	intervalle	§5.7
`/`		190	G	division entière	§5.3
`/:`	$\notin$	160	G	non-appartenance	§4.4
`/<:`	$\not\subseteq$	110	G	non-inclusion	§4.5
`/<<:`	$\not\subset$	110	G	non-inclusion stricte	§4.5
`/=`	$\neq$	160	G	inégalité	§4.3
`/\`	$\cap$	160	G	intersection	§5.8
`/|\`	$\uparrow$	160	G	restriction d'une suite à la tête	§5.19
`:`	$\in$	120	G	appartenance	§4.4
`:`		60	G	champ de record	§5.9
`::`	$:\in$		G	devient élément de	§6.12
`:=`			G	devient égal	§6.3
`;`		20	G	séquencement de substitutions ou composition de relations	§6.15, §5.11
`<`		160	G	strictement inférieur ou délimiteur de fichier de définitions	§4.6, §2.3
`<+`	$\lhd\!\!\!+$	160	G	surcharge d'une relation	§5.14
`<->`	$\leftrightarrow$	125	G	ensemble des relations	§5.10
`<-`	$\leftarrow$	160	G	insertion en fin de suite	§5.19
`<--`	$\longleftarrow$		G	paramètres de sortie d'opération	§6.16, §7.23
`<:`	$\subseteq$	110	G	inclusion	§4.5
`<<:`	$\subset$	110	G	inclusion stricte	§4.5
`<<|`	$\lhd\!\!\!-$	160	G	soustraction sur le domaine	§5.14
`<=`	$\leq$	160	G	inférieur ou égal	§4.6
`<=>`	$\Leftrightarrow$	60	G	équivalence	§4.1
`<|`	$\lhd$	160	G	restriction sur le domaine	§5.14
`=`		60	G	égalité	§4.3
`==`				définition	§2.3
`=>`	$\Rightarrow$	30	G	implique	§4.1
`>`		160	G	strictement supérieur ou délimiteur de fichier de définitions	§4.6, §2.3
`>+>`	$\rightarrowtail\!\!\!\!\!+$	125	G	injection partielle	§5.15
`>->`	$\rightarrowtail$	125	G	injection totale	§5.15
`>->>`	$\rightarrowtail\!\!\!\!\!\twoheadrightarrow$	125	G	bijection totale	§5.15
`><`	$\otimes$	160	G	produit direct de relations	§5.11

ASCII	**Math.**	**Pri.**	**As.**	**Description**	**Référence**
>=	$\geq$	160	G	supérieur ou égal	§4.6
ABSTRACT_CONSTANTS				clause ABSTRACT_CONSTANTS	§7.15
ABSTRACT_VARIABLES				clause ABSTRACT_VARIABLES	§7.19
ANY				substitution ANY	§6.10
ASSERT				substitution ASSERT	§6.5
ASSERTIONS				clause ASSERTIONS	§7.21
BE				substitution LET	§6.11
BEGIN				substitution BEGIN	§6.1
BOOL				ensemble des booléens	§5.6
CASE				substitution CASE	§6.9
CHOICE				substitution CHOICE	§6.6
CONCRETE_CONSTANTS				clause CONCRETE_CONSTANTS	§7.14
CONCRETE_VARIABLES				clause CONCRETE_VARIABLES	§7.18
CONSTANTS				clause CONSTANTS	§7.14
CONSTRAINTS				clause CONSTRAINTS	§7.5
DEFINITIONS				clause DEFINITIONS	§2.3
DO				substitution WHILE	§6.17
EITHER				substitution CASE	§6.9
ELSE				substitution IF, SELECT ou CASE	§6.7, §6.8, §6.9
ELSIF				substitution IF	§6.7
END				terminateur des clauses ou des substitutions BEGIN, PRE, ASSERT, CHOICE, IF, SELECT, ANY, LET, VAR, CASE et WHILE	
EXTENDS				clause EXTENDS	§7.11
FALSE				constante booléenne littérale “faux”	§5.2
FIN	$\mathbb{F}$			ensemble des sous-ensembles finis	§5.7
FIN1	$\mathbb{F}_1$			ensemble des sous-ensembles finis non-vides	§5.7
IF				substitution IF	§6.7
IMPLEMENTATION				clause IMPLEMENTATION	§7.4
IMPORTS				clause IMPORTS	§7.7
IN				substitution LET ou VAR	§6.11, §6.14
INCLUDES				clause INCLUDES	§7.9
INITIALISATION				clause INITIALISATION	§7.22
INT				ensemble des entiers relatifs concrets	§5.6
INTEGER	$\mathbb{Z}$			ensemble des entiers relatifs	§5.6
INTER	$\bigcap$			intersection quantifiée	§5.8
INVARIANT				clause INVARIANT ou substitution WHILE	§7.20, §6.17
LET				substitution LET	§6.11
LOCAL_OPERATIONS				clause LOCAL_OPERATIONS	§7.24
MACHINE				clause MACHINE	§7.1
MAXINT				plus grand entier implémentable	§5.3

ASCII	Math.	Pri.	As.	Description	Référence
MININT				plus petit entier implémentable	§5.3
NAT				ensemble des entiers naturels concrets	§5.6
NAT1	NAT_1			ensemble des entiers naturels non nuls concrets	§5.6
NATURAL	$\mathbb{N}$			ensemble des entiers naturels	§5.6
NATURAL1	$\mathbb{N}_1$			ensemble des entiers naturels non nuls	§5.6
OF				substitution CASE	§6.9
OPERATIONS				clause OPERATIONS	§7.23
OR				substitution CHOICE ou CASE	§6.6, §6.9
PI	Π			produit quantifié d'entiers	§5.4
POW	$\mathbb{P}$			ensemble des sous-ensembles	§5.7
POW1	$\mathbb{P}_1$			ensemble des sous-ensembles non vides	§5.7
PRE				substitution précondition	§6.4
PROMOTES				clause PROMOTES	§7.10
PROPERTIES				clause PROPERTIES	§7.16
REFINES				clause REFINES	§7.6
REFINEMENT				clause REFINEMENT	§7.3
SEES				clause SEES	§7.8
SELECT				substitution SELECT	§6.8
SETS				clause SETS	§7.13
SIGMA	Σ			somme quantifiée	§5.4
STRING				ensemble des chaînes de caractères	§5.6
THEN				substitution précondition, ASSERT, IF, CASE ou SELECT	§6.4, §6.5, §6.7, §6.9, §6.8
TRUE				constante booléenne littérale "vrai"	§5.2
UNION	$\bigcup$			union quantifiée	§5.8
USES				clause USES	§7.12
VALUES				clause VALUES	§7.17
VAR				substitution VAR	§6.14
VARIANT				substitution WHILE	§6.17
VARIABLES				clause VARIABLES	§7.19
WHEN				substitution SELECT	§6.8
WHERE				substitution ANY	§6.10
WHILE				substitution WHILE	§6.17
[				image, début de suite	§5.13, §5.17
[]				suite vide	§5.17
\/	$\cup$	160	G	union	§5.8
|/	$\downarrow$	160	G	restriction d'une suite à la queue	§5.19
]				image, fin de suite	§5.13, §5.17
^	$\hat{}$	160	G	concaténation de suites	§5.19
arity				arité du nœud d'un arbre	§5.22
bin				arbre binaire en extension	§5.23

ASCII	**Math.**	**Pri.**	**As.**	**Description**	**Référence**
`bool`				conversion d'un prédicat en booléen	§5.2
`btree`				arbres binaires	§5.20
`card`				cardinal	§5.4
`closure(R)`	R^{*}			fermeture réflexive d'une relation	§5.12
`closure1(R)`	R^{+}			fermeture d'une relation	§5.12
`conc`				concaténation de suites	§5.19
`const`				construction d'un arbre	§5.21
`dom`				domaine d'une fonction	§5.13
`father`				père du nœud d'un arbre	§5.22
`first`				premier élément d'une suite	§5.18
`fnc`				transformée en fonction	§5.16
`front`				tête d'une suite	§5.18
`id`				fonction identité	§5.11
`infix`				aplatissement infixé d'un arbre	§5.23
`inter`				intersection généralisée	§5.8
`iseq`				ensemble des suites injectives	§5.17
`iseq1`	iseq_1			ensemble des suites injectives non-vides	§5.17
`iterate(R, n)`	R^{n}			itération d'une relation	§5.12
`last`				dernier élément d'une suite	§5.18
`left`				sous-arbre gauche	§5.23
`max`				maximum d'un ensemble d'entiers	§5.4
`min`				minimum d'un ensemble d'entiers	§5.4
`mirror`				symétrie d'un arbre	§5.21
`mod`		190	G	modulo	§5.3
`not`	$\neg$			négation (NON logique)	§4.1
`or`	$\vee$	40	G	disjonction (OU logique)	§4.1
`perm`				ensemble des permutations (suites bijectives)	§5.17
`postfix`				aplatissement postfixé d'un arbre	§5.21
`pred`				prédécesseur d'un entier	§5.3
`prefix`				aplatissement préfixé d'un arbre	§5.21
`prj1`	prj_1			première projection d'une relation	§5.11
`prj2`	prj_2			seconde projection d'une relation	§5.11
`ran`				codomaine d'une relation	§5.13
`rank`				rang du nœud d'un arbre	§5.22
`rec`				record en extension	§5.9
`rel`				transformée en relation	§5.16
`rev`				inverse d'une suite	§5.18
`right`				sous-arbre droit	§5.23
`seq`				ensemble des suites	§5.17
`seq1`				ensemble des suites non-vides	§5.17
`size`				taille d'une suite	§5.18

<table>
<tr><th>ASCII</th><th>Math.</th><th>Pri.</th><th>As.</th><th>Description</th><th>Référence</th></tr>
<tr><td>sizet</td><td></td><td></td><td></td><td>taille d’un arbre</td><td>§5.21</td></tr>
<tr><td>skip</td><td></td><td></td><td></td><td>substitution identité</td><td>§6.2</td></tr>
<tr><td>son</td><td></td><td></td><td></td><td>$i^{\text{ème}}$ fils du nœud d’un arbre</td><td>§5.22</td></tr>
<tr><td>sons</td><td></td><td></td><td></td><td>fils du nœud d’un arbre</td><td>§5.21</td></tr>
<tr><td>struct</td><td></td><td></td><td></td><td>ensemble de records</td><td>§5.9</td></tr>
<tr><td>subtree</td><td></td><td></td><td></td><td>sous-arbre d’un arbre</td><td>§5.22</td></tr>
<tr><td>succ</td><td></td><td></td><td></td><td>successeur</td><td>§5.3</td></tr>
<tr><td>tail</td><td></td><td></td><td></td><td>queue d’une suite</td><td>§5.18</td></tr>
<tr><td>top</td><td></td><td></td><td></td><td>racine d’un arbre</td><td>§5.21</td></tr>
<tr><td>tree</td><td></td><td></td><td></td><td>arbres</td><td>§5.20</td></tr>
<tr><td>union</td><td></td><td></td><td></td><td>union généralisée</td><td>§5.8</td></tr>
<tr><td>{</td><td></td><td></td><td></td><td>début d’ensemble</td><td>§5.7</td></tr>
<tr><td>{}</td><td>Ø</td><td></td><td></td><td>ensemble vide</td><td>§5.6</td></tr>
<tr><td>|</td><td></td><td>10</td><td>G</td><td>barre verticale utilisée dans ∀, ∃, ∪, ∩, Σ, Π, λ, { | }</td><td></td></tr>
<tr><td>|-></td><td>↦</td><td>160</td><td>G</td><td>maplet</td><td>§5.5</td></tr>
<tr><td>|></td><td>▷</td><td>160</td><td>G</td><td>restriction sur le codomaine</td><td>§5.14</td></tr>
<tr><td>|>></td><td>⩥</td><td>160</td><td>G</td><td>soustraction sur le codomaine</td><td>§5.14</td></tr>
<tr><td>||</td><td></td><td>20</td><td>G</td><td>substitutions simultanées ou produit parallèle de relations</td><td>§6.18, §5.11</td></tr>
<tr><td>}</td><td></td><td></td><td></td><td>fin d’ensemble</td><td>§5.7</td></tr>
<tr><td>*r*~</td><td>r^{-1}</td><td>230</td><td>G</td><td>relation inverse</td><td>§5.11</td></tr>
</table>

ANNEXE B GRAMMAIRES

Nous regroupons dans cette annexe, la grammaire du langage B, la grammaire des prédicats de typage et la grammaire des types. Les conventions lexicales et syntaxiques utilisées pour décrire ces grammaires sont définies au chapitre 2.

B.1 Grammaire du langage B

B.1.1 Axiome

Composant ::=
    Machine_abstraite
| *Raffinement*
| *Implantation*

B.1.2 Clauses

Machine_abstraite ::=
    "MACHINE" *En-tête*
    Clause_machine_abstraite*
    "END"

Clause_machine_abstraite ::=
    Clause_constraints
| *Clause_sees*
| *Clause_includes*
| *Clause_promotes*
| *Clause_extends*
| *Clause_uses*
| *Clause_sets*
| *Clause_concrete_constants*
| *Clause_abstract_constants*
| *Clause_properties*
| *Clause_concrete_variables*
| *Clause_abstract_variables*
| *Clause_invariant*
| *Clause_assertions*
| *Clause_initialisation*
| *Clause_operations*

En-tête ::=
    Ident ["(" Ident$^{+\text{","}}$ ")"]

Raffinement ::=
    "REFINEMENT" *En-tête*
    Clause_refines
    Clause_raffinement*
    "END"

Clause_raffinement ::=
  Clause_sees
| *Clause_includes*
| *Clause_promotes*
| *Clause_extends*
| *Clause_sets*
| *Clause_concrete_constants*
| *Clause_abstract_constants*
| *Clause_properties*
| *Clause_concrete_variables*
| *Clause_abstract_variables*
| *Clause_invariant*
| *Clause_assertions*
| *Clause_initialisation*
| *Clause_operations*

Implantation ::=
  "IMPLEMENTATION" *En-tête*
  Clause_refines
  *Clause_implantation**
  "END"

Clause_implantation ::=
  Clause_sees
| *Clause_imports*
| *Clause_promotes*
| *Clause_extends_B0*
| *Clause_sets*
| *Clause_concrete_constants*
| *Clause_properties*
| *Clause_values*
| *Clause_concrete_variables*
| *Clause_invariant*
| *Clause_assertions*
| *Clause_initialisation_B0*
| *Clause_operations_B0*
| *Clause_operations_locales*

Clause_constraints ::=
  "CONSTRAINTS" *Prédicat*

Clause_refines ::=
  "REFINES" Ident

Clause_IMPORTS ::=
  "IMPORTS" (*Ident_ren* ["(" *Instanciation_B0*$^{+","}$ ")"])$^{+","}$

Instanciation_B0 :=
  Terme
| *Ensemble_nombres_B0*
| "BOOL"
| *Intervalle_B0*

Clause_sees ::=
  "SEES" *Ident_ren*$^{+","}$

Clause_includes ::=
  "INCLUDES" (*Ident_ren* ["(" *Instanciation*$^{+","}$ ")"])$^{+","}$

Instanciation :=
  Terme
| *Ensemble_nombres*
| "BOOL"
| *Intervalle*

Clause_promotes ::=
    "PROMOTES" *Ident_ren*$^{+","}$

Clause_EXTENDS ::=
    "EXTENDS" (*Ident_ren* ["(" *Instanciation*$^{+","}$ ")"])$^{+","}$

Clause_EXTENDS_B0 ::=
    "EXTENDS" (*Ident_ren* ["(" *Instanciation_B0*$^{+","}$ ")"])$^{+","}$

Clause_uses ::=
    "USES" *Ident_ren*$^{+","}$

Clause_sets ::=
    "SETS" *Ensemble*$^{+";"}$

Ensemble ::=
    Ident
| Ident "=" "{" Ident$^{+","}$ "}"

Clause_concrete_constants ::=
    "CONCRETE_CONSTANTS" Ident$^{+","}$
| "CONSTANTS" Ident$^{+","}$

Clause_abstract_constants ::=
    "ABSTRACT_CONSTANTS" Ident$^{+","}$

Clause_properties ::=
    "PROPERTIES" *Prédicat*

Clause_values ::=
    "VALUES" *Valuation*$^{+";"}$

Valuation ::=
    Ident "=" *Terme*
| Ident "=" *Expr_tableau*
| Ident "=" *Intervalle_B0*

Clause_concrete_variables ::=
    "CONCRETE_VARIABLES" *Ident_ren*$^{+","}$

Clause_abstract_variables ::=
    "ABSTRACT_VARIABLES" *Ident_ren*$^{+","}$
| "VARIABLES" *Ident_ren*$^{+","}$

Clause_invariant ::=
    "INVARIANT" *Prédicat*

Clause_assertions ::=
    "ASSERTIONS" *Prédicat*$^{+";"}$

Clause_initialisation ::=
    "INITIALISATION" *Substitution*

Clause_initialisation_B0 ::=
    "INITIALISATION" *Instruction*

Clause_operations ::=
    "OPERATIONS" *Opération*$^{+";"}$

Opération ::=
    Entête_opération "=" *Substitution_corps_opération*

Entête_opération ::=
    [Ident$^{+","}$ "←"] *Ident_ren* ["(" Ident$^{+","}$ ")"]

Clause_operations_B0 ::=
    "OPERATIONS" *Opération_B0*$^{+";"}$

Opération_B0 ::=
Entête_opération "=" *Instruction_corps_opération*

Clause_operations_locales ::=
"LOCAL_OPERATIONS" $Opération^{+";"}$

B.1.3 Termes et regroupement d'expressions

Terme ::= *Terme_simple*
| *Expression_arithmétique*
| *Terme_record*
| *Terme_record* ("'" Ident)$^{+}$

Terme_simple ::=
Ident_ren
| *Entier_lit*
| *Booléen_lit*
| "bool" "(" *Condition* ")"
| *Ident_ren* ("'" Ident)$^{+}$

Entier_lit ::= Entier_littéral
| "MAXINT"
| "MININT"

Booléen_lit ::= "FALSE"
| "TRUE"

Expression_arithmétique ::=
Entier_lit
| *Ident_ren*
| *Ident_ren* "(" $Terme^{+","}$ ")"
| *Ident_ren* ("'" Ident)$^{+}$
| *Expression_arithmétique* "+" *Expression_arithmétique*
| *Expression_arithmétique* "-" *Expression_arithmétique*
| "-" *Expression_arithmétique*
| *Expression_arithmétique* "×" *Expression_arithmétique*
| *Expression_arithmétique* "/" *Expression_arithmétique*
| *Expression_arithmétique* "mod" *Expression_arithmétique*
| $Expression\_arithmétique^{Expression\_arithmétique}$
| "succ" "(" *Expression_arithmétique* ")"
| "pred" "(" *Expression_arithmétique* ")"
| "floor" "(" *Expression_arithmétique* ")"
| "ceiling" "(" *Expression_arithmétique* ")"
| "real" "(" *Expression_arithmétique* ")"
| "(" *Expression_arithmétique* ")"

Terme_record ::=
"rec" "(" ([Ident ":"] (*Terme* | *Expr_tableau*))$^{+","}$ ")"

Expr_tableau ::=
Ident
| "{" ($Terme\_simple^{+"\mapsto"}$ "↦" *Terme*)$^{+","}$ "}"
| $Ensemble\_simple^{+"\times"}$ "×" "{" *Terme* "}"

Intervalle_B0 ::=
Expression_arithmétique ".." *Expression_arithmétique*
| *Ensemble_nombres_B0*

Ensemble_nombres_B0 ::=
"NAT"
| "NAT_1"
| "INT"

B.1.4 Conditions

Condition ::=
Terme_simple "=" *Terme_simple*
| *Terme_simple* "≠" *Terme_simple*
| *Terme_simple* "<" *Terme_simple*
| *Terme_simple* ">" *Terme_simple*
| *Terme_simple* "≤" *Terme_simple*
| *Terme_simple* "≥" *Terme_simple*
| *Condition* "∧" *Condition*
| *Condition* "∨" *Condition*
| "¬" "(" *Condition* ")"
| "(" *Condition* ")"

B.1.5 Instructions

Instruction ::=
Instruction_bloc
| *Instruction_variable_locale*
| *Substitution_identité*
| *Instruction_devient_égal*
| *Instruction_appel_opération*
| *Instruction_conditionnelle*
| *Instruction_cas*
| *Instruction_assertion*
| *Instruction_séquence*
| *Substitution_tant_que*

Instruction_corps_opération ::=
Instruction_bloc
| *Instruction_variable_locale*
| *Substitution_identité*
| *Instruction_devient_égal*
| *Instruction_appel_opération*
| *Instruction_conditionnelle*
| *Instruction_cas*
| *Instruction_assertion*
| *Substitution_tant_que*

Instruction_bloc ::=
"BEGIN" *Instruction* "END"

Instruction_variable_locale ::=
"VAR" Ident$^{+","}$ "IN" *Instruction* "END"

Instruction_devient_égal ::=
Ident_ren ["(" *Terme*$^{+","}$ ")"] ":=" *Terme*
| *Ident_ren* ":=" *Expr_tableau*
| *Ident_ren* ("'" Ident)$^{+}$ ":=" *Terme*

Instruction_appel_opération ::=
[*Ident_ren*$^{+","}$ "⟵"] *Ident_ren* ["(" (*Terme* | Chaîne_lit)$^{+","}$ ")"]

Instruction_séquence ::=
    Instruction ";" *Instruction*

Instruction_conditionnelle ::=
    "IF" *Condition* "THEN" *Instruction*
    ("ELSIF" *Condition* "THEN" *Instruction*)*
    ["ELSE" *Instruction*]
    "END"

Instruction_cas ::=
    "CASE" *Terme_simple* "OF"
    "EITHER" *Terme_simple*$^{+","}$ "THEN" *Instruction*
    ("OR" *Terme_simple*$^{+","}$ "THEN" *Instruction*)*
    ["ELSE" *Instruction*]
    "END"
    "END"

Substitution_tant_que ::=
    "WHILE" *Condition* "DO" *Instruction*
    "INVARIANT" *Prédicat*
    "VARIANT" *Expression*
    "END"

Instruction_assertion ::=
    "ASSERT" *Condition* "THEN" *Instruction* "END"

B.1.6 Prédicats

Prédicat ::=
    Prédicat_parenthésé
| *Prédicat_conjonction*
| *Prédicat_négation*
| *Prédicat_disjonction*
| *Prédicat_implication*
| *Prédicat_équivalence*
| *Prédicat_universel*
| *Prédicat_existentiel*
| *Prédicat_égalité*
| *Prédicat_inégalité*
| *Prédicat_appartenance*
| *Prédicat_non_appartenance*
| *Prédicat_inclusion*
| *Prédicat_inclusion_stricte*
| *Prédicat_non_inclusion*
| *Prédicat_non_inclusion_stricte*
| *Prédicat_inférieur_ou_égal*
| *Prédicat_strictement_inférieur*
| *Prédicat_supérieur_ou_égal*
| *Prédicat_strictement_supérieur*

Prédicat_parenthésé ::= "(" *Prédicat* ")"

Prédicat_conjonction ::= *Prédicat* "∧" *Prédicat*

Prédicat_négation ::= "¬" "(" *Prédicat* ")"

Prédicat_disjonction ::= *Prédicat* "∨" *Prédicat*

Prédicat_implication ::= *Prédicat* "⇒" *Prédicat*

Prédicat_équivalence ::= *Prédicat* "⇔" *Prédicat*

Prédicat_universel ::= "∀" *Liste_ident* "." "(" *Prédicat* "⇒" *Prédicat* ")"

Prédicat_existentiel	::=	"∃" *Liste_ident* "." "(" *Prédicat* ")"
Prédicat_égalité	::=	*Expression* "=" *Expression*
Prédicat_inégalité	::=	*Expression* "≠" *Expression*
Prédicat_appartenance	::=	*Expression* "∈" *Expression*
Prédicat_non_appartenance	::=	*Expression* "∉" *Expression*
Prédicat_inclusion	::=	*Expression* "⊆" *Expression*
Prédicat_inclusion_stricte	::=	*Expression* "⊂" *Expression*
Prédicat_non_inclusion	::=	*Expression* "⊈" *Expression*
Prédicat_non_inclusion_stricte	::=	*Expression* "⊄" *Expression*
Prédicat_inférieur_ou_égal	::=	*Expression* "≤" *Expression*
Prédicat_strictement_inférieur	::=	*Expression* "<" *Expression*
Prédicat_supérieur_ou_égal	::=	*Expression* "≥" *Expression*
Prédicat_strictement_supérieur	::=	*Expression* ">" *Expression*

B.1.7 Expressions

Expression ::=
Expression_primaire
| *Expression_booléenne*
| *Expression_arithmétique*
| *Expression_de_couples*
| *Expression_d_ensembles*
| *Construction_d_ensembles*
| *Expression_de_records*
| *Expression_de_relations*
| *Expression_de_fonctions*
| *Construction_de_fonctions*
| *Expression_de_suites*
| *Construction_de_suites*
| *Expression_d_arbres*

Expression_primaire ::=
Donnée
| *Expr_parenthésée*
| *Chaîne_lit*

Expression_booléenne ::=
Booléen_lit
| *Conversion_bool*

```
Expression_arithmétique ::=
            Entier_lit
  |         Addition
  |         Différence
  |         Moins_unaire
  |         Produit
  |         Division
  |         Modulo
  |         Puissance
  |         Successeur
  |         Prédécesseur
  |         Maximum
  |         Minimum
  |         Cardinal
  |         Somme_généralisée
  |         Produit_généralisé
  |         Partie_entière
  |         Partie_entière_par_excès
  |         Conversion_ℤ⊄ℝ

Expression_de_couples ::=
            Couple

Expression_d_ensembles ::=
            Ensemble_vide
  |         Ensemble_nombres
  |         Ensemble_booléen
  |         Ensemble_chaînes

Construction_d_ensembles ::=
            Produit
  |         Ens_compréhension
  |         Sous_ensembles
  |         Sous_ensembles_finis
  |         Ens_extension
  |         Intervalle
  |         Différence
  |         Union
  |         Intersection
  |         Union_généralisée
  |         Intersection_généralisée
  |         Union_quantifiée
  |         Intersection_quantifée

Expression_de_records ::=
            Ensemble_records
  |         Record_en_extension
  |         Champ_de_record
```

Expression_de_relations ::=
Ensemble_relations
| *Identité*
| *Inverse*
| *Première_projection*
| *Deuxième_projection*
| *Composition*
| *Produit_directe*
| *Produit_parallèle*
| *Itération*
| *Fermeture_réflexive*
| *Fermeture*
| *Domaine*
| *Codomaine*
| *Image*
| *Restriction_domaine*
| *Soustraction_domaine*
| *Restriction_codomaine*
| *Soustraction_codomaine*
| *Surcharge*

Expression_de_fonctions ::=
Fonction_partielle
| *Fonction_totale*
| *Injection_partielle*
| *Injection_totale*
| *Surjection_partielle*
| *Surjection_totale*
| *Bijection_totale*

Construction_de_fonctions ::=
Lambda_expression
| *Évaluation_fonction*
| *Transformée_fonction*
| *Transformée_relation*

Expression_de_suites ::=
Suites
| *Suites_non_vide*
| *Suites_injectives*
| *Suites_inj_non_vide*
| *Permutations*
| *Suite_vide*
| *Suite_extension*

Construction_de_suites ::=
Taille_suite
| *Premier_élément_suite*
| *Dernier_élément_suite*
| *Tête_suite*
| *Queue_suite*
| *Inverse_suite*
| *Concaténation*
| *Insertion_tête*
| *Insertion_queue*
| *Restriction_tête*
| *Restriction_queue*
| *Concat_généralisée*

Expression_d_arbres ::=
Arbres
| *Arbres_binaires*
| *Construction_arbre*
| *Racine_arbre*
| *Fils_arbre*
| *Aplatissement_préfixé*
| *Aplatissement_postfixé*
| *Taille_arbre*
| *Symétrie_arbre*
| *Rang_noeud*
| *Père_noeud*
| *Fils_noeud*
| *Sous_arbre_noeud*
| *Arité_noeud*

Donnée ::= *Ident_ren*
| *Ident_ren*"$0"

Expr_parenthésée ::= "(" *Expression* ")"

Chaîne_lit ::= Chaîne_de_caractères

Booléen_lit ::= "FALSE"
| "TRUE"

Conversion_bool ::= "bool" "(" *Prédicat* ")"

Entier_lit ::= Entier_littéral
| "MAXINT"
| "MININT"

Addition ::= *Expression* "+" *Expression*

Différence ::= *Expression* "-" *Expression*

Moins_unaire ::= "-" *Expression*

Produit ::= *Expression* "×" *Expression*

Division ::= *Expression* "/" *Expression*

Modulo ::= *Expression* "mod" *Expression*

Puissance ::= $\mathit{Expression}^{\mathit{Expression}}$

Successeur ::= "succ" ["(" *Expression* ")"]

Prédécesseur ::= "pred" ["(" *Expression* ")"]

Maximum ::= "max" "(" *Expression* ")"

Minimum ::= "min" "(" *Expression* ")"

Cardinal ::= "card" "(" *Expression* ")"

Partie_entière ::= "floor" "(" *Expression* ")"

Partie_entière_par_excès ::= "ceiling" "(" *Expression* ")"

Conversion_ℤ⊄ℝ ::= "real" "(" *Expression* ")"

Somme_généralisée ::= "Σ" *Liste_ident* "." "(" *Prédicat* "|" *Expression* ")"

Produit_généralisé ::= "Π" Liste_ident "." "(" *Prédicat* "|" *Expression* ")"

Couple ::= *Expression* "↦" *Expression*
| *Expression* "," *Expression*

Ensemble_vide ::= "∅"

Ensemble_nombres ::= "ℤ"
| "ℕ"

<table>
<tr><td></td><td>|</td><td>"$\mathbb{N}_1$"</td></tr>
<tr><td></td><td>|</td><td>"NAT"</td></tr>
<tr><td></td><td>|</td><td>"NAT_1"</td></tr>
<tr><td></td><td>|</td><td>"INT"</td></tr>
<tr><td>Ensemble_booléen</td><td>::=</td><td>"BOOL"</td></tr>
<tr><td>Ensemble_chaînes</td><td>::=</td><td>"STRING"</td></tr>
<tr><td>Ens_compréhension</td><td>::=</td><td>"{" Ident$^{+","}$ "|" Prédicat "}"</td></tr>
<tr><td>Sous_ensembles</td><td>::=</td><td>"$\mathbb{P}$" "(" Expression ")"</td></tr>
<tr><td></td><td>|</td><td>"$\mathbb{P}_1$" "(" Expression ")"</td></tr>
<tr><td>Sous_ensembles_finis</td><td>::=</td><td>"$\mathbb{F}$" "(" Expression ")"</td></tr>
<tr><td></td><td>|</td><td>"$\mathbb{F}_1$" "(" Expression ")"</td></tr>
<tr><td>Ens_extension</td><td>::=</td><td>"{" Expression$^{+","}$ "}"</td></tr>
<tr><td>Intervalle</td><td>::=</td><td>Expression ".." Expression</td></tr>
<tr><td>Union</td><td>::=</td><td>Expression "∪" Expression</td></tr>
<tr><td>Intersection</td><td>::=</td><td>Expression "∩" Expression</td></tr>
<tr><td>Union_généralisée</td><td>::=</td><td>"union" "(" Expression ")"</td></tr>
<tr><td>Intersection_généralisée</td><td>::=</td><td>"inter" "(" Expression ")"</td></tr>
<tr><td>Union_quantifiée</td><td>::=</td><td>"⋃" Liste_ident "." "(" Prédicat "|" Expression ")"</td></tr>
<tr><td>Intersection_quantifée</td><td>::=</td><td>"⋂" Liste_ident "." "(" Prédicat "|" Expression ")"</td></tr>
<tr><td>Ensemble_records</td><td>::=</td><td>"struct" "(" (Ident ":" Expression)$^{+","}$ ")"</td></tr>
<tr><td>Record_en_extension</td><td>::=</td><td>"rec" "(" ([Ident ":"] Expression)$^{+","}$ ")"</td></tr>
<tr><td>Champ_de_record</td><td>::=</td><td>Expression "'" Ident</td></tr>
<tr><td>Ensemble_relations</td><td>::=</td><td>Expression "↔" Expression</td></tr>
<tr><td>Identité</td><td>::=</td><td>"id" "(" Expression ")"</td></tr>
<tr><td>Inverse</td><td>::=</td><td>Expression "$^{-1}$"</td></tr>
<tr><td>Première_projection</td><td>::=</td><td>"prj_1" "(" Expression "," Expression ")"</td></tr>
<tr><td>Deuxième_projection</td><td>::=</td><td>"prj_2" "(" Expression "," Expression ")"</td></tr>
<tr><td>Composition</td><td>::=</td><td>Expression ";" Expression</td></tr>
<tr><td>Produit_direct</td><td>::=</td><td>Expression "⊗" Expression</td></tr>
<tr><td>Produit_parallèle</td><td>::=</td><td>Expression "||" Expression</td></tr>
<tr><td>Itération</td><td>::=</td><td>$\mathit{Expression}^{\mathit{Expression}}$</td></tr>
<tr><td>Fermeture_réflexive</td><td>::=</td><td>Expression$^{"*"}$</td></tr>
<tr><td>Fermeture</td><td>::=</td><td>Expression$^{"+"}$</td></tr>
<tr><td>Domaine</td><td>::=</td><td>"dom" "(" Expression ")"</td></tr>
<tr><td>Codomaine</td><td>::=</td><td>"ran" "(" Expression ")"</td></tr>
<tr><td>Image</td><td>::=</td><td>Expression "[" Expression "]"</td></tr>
<tr><td>Restriction_domaine</td><td>::=</td><td>Expression "◁" Expression</td></tr>
<tr><td>Soustraction_domaine</td><td>::=</td><td>Expression "⩤" Expression</td></tr>
<tr><td>Restriction_codomaine</td><td>::=</td><td>Expression "▷" Expression</td></tr>
<tr><td>Soustraction_codomaine</td><td>::=</td><td>Expression "⩥" Expression</td></tr>
<tr><td>Surcharge</td><td>::=</td><td>Expression "<+" Expression</td></tr>
<tr><td>Fonction_partielle</td><td>::=</td><td>Expression "⇸" Expression</td></tr>
</table>

Fonction_totale	::=	*Expression* "→" *Expression*
Injection_partielle	::=	*Expression* "⤔" *Expression*
Injection_totale	::=	*Expression* "↣" *Expression*
Surjection_partielle	::=	*Expression* "⤀" *Expression*
Surjection_totale	::=	*Expression* "↠" *Expression*
Bijection_totale	::=	*Expression* "⤖" *Expression*
Lambda_expression	::=	"λ" *Liste_ident* "." "(" *Prédicat* "|" *Expression* ")"
Évaluation_fonction	::=	*Expression* "(" *Expression* ")"
Transformée_fonction	::=	"fnc" "(" *Expression* ")"
Transformée_relation	::=	"rel" "(" *Expression* ")"
Suites	::=	"seq" "(" *Expression* ")"
Suites_non_vide	::=	"seq_1" "(" *Expression* ")"
Suites_injectives	::=	"iseq" "(" *Expression* ")"
Suites_inj_non_vide	::=	"$iseq_1$" "(" *Expression* ")"
Permutations	::=	"perm" "(" *Expression* ")"
Suite_vide	::=	"[]"
Suite_extension	::=	"[" *Expression*$^{+","}$ "]"
Taille_suite	::=	"size" "(" *Expression* ")"
Premier_élément_suite	::=	"first" "(" *Expression* ")"
Dernier_élément_suite	::=	"last" "(" *Expression* ")"
Tête_suite	::=	"front" "(" *Expression* ")"
Queue_suite	::=	"tail" "(" *Expression* ")"
Inverse_suite	::=	"rev" "(" *Expression* ")"
Concaténation	::=	*Expression* "^" *Expression*
Insertion_tête	::=	*Expression* "→" *Expression*
Insertion_queue	::=	*Expression* "←" *Expression*
Restriction_tête	::=	*Expression* "↑" *Expression*
Restriction_queue	::=	*Expression* "↓" *Expression*
Concat_généralisée	::=	"conc" "(" *Expression* ")"
Arbres	::=	"tree" "(" *Expression* ")"
Arbres_binaires	::=	"btree" "(" *Expression* ")"
Construction_arbre	::=	"const" "(" *Expression* "," *Expression* ")"
Racine_arbre	::=	"top" "(" *Expression* ")"
Fils_arbre	::=	"sons" "(" *Expression* ")"
Aplatissement_préfixé	::=	"prefix" "(" *Expression* ")"
Aplatissement_postfixé	::=	"postfix" "(" *Expression* ")"
Taille_arbre	::=	"sizet" "(" *Expression* ")"
Symétrie_arbre	::=	"mirror" "(" *Expression* ")"
Rang_noeud	::=	"rank" "(" *Expression* "," *Expression* ")"
Père_noeud	::=	"father" "(" *Expression* "," *Expression* ")"
Fils_noeud	::=	"son" "(" *Expression* "," *Expression* "," *Expression* ")"

Sous_arbre_noeud ::= "subtree" (" *Expression* "," *Expression* ")"

Arité_noeud ::= "arity" "(" *Expression* "," *Expression* ")"

Arbre_binaire_en_extension ::= "bin" "(" *Expression* [“,” *Expression* “,” *Expression*] ")"

Sous_arbre_gauche ::= "left" "(" *Expression* ")"

Sous_arbre_droit ::= "right" "(" *Expression* ")"

Aplatissement_infixé ::= "infix" "(" *Expression* ")"

B.1.8 Substitutions

Substitution ::=
    Substitution_bloc
  | *Substitution_identité*
  | *Substitution_devient_égal*
  | *Substitution_précondition*
  | *Substitution_assertion*
  | *Substitution_choix_borné*
  | *Substitution_conditionnelle*
  | *Substitution_sélection*
  | *Substitution_cas*
  | *Substitution_choix_non_borné*
  | *Substitution_définition_locale*
  | *Substitution_devient_elt_de*
  | *Substitution_devient_tel_que*
  | *Substitution_variable_locale*
  | *Substitution_séquence*
  | *Substitution_appel_opération*
  | *Substitution_simultanée*
  | *Substitution_tant_que*

Substitution_corps_opération ::=
    Substitution_bloc
  | *Substitution_identité*
  | *Substitution_devient_égal*
  | *Substitution_précondition*
  | *Substitution_assertion*
  | *Substitution_choix_borné*
  | *Substitution_conditionnelle*
  | *Substitution_sélection*
  | *Substitution_cas*
  | *Substitution_any*
  | *Substitution_let*
  | *Substitution_devient_elt_de*
  | *Substitution_devient_tel_que*
  | *Substitution_variable_locale*
  | *Substitution_appel_opération*

Substitution_bloc ::=
    "BEGIN" *Substitution* "END"

Substitution_identité ::=
    "skip"

Substitution_devient_égal ::=
    Ident_ren$^{+\text{","}}$ ":=" *Expression*$^{+\text{","}}$
  | *Ident_ren* "(" *Expression*$^{+\text{","}}$ ")" ":=" *Expression*
  | *Ident_ren* ("'" Ident)$^{+}$ ":=" *Expression*

Substitution_précondition ::=

"PRE" *Prédicat* "THEN" *Substitution* "END"

Substitution_assertion ::=

"ASSERT" *Prédicat* "THEN" *Substitution* "END"

Substitution_choix_borné ::=

"CHOICE" *Substitution* ("OR" *Substitution*)* "END"

Substitution_conditionnelle ::=
"IF" *Prédicat* "THEN" *Substitution*
("ELSIF" *Prédicat* "THEN" *Substitution*)*
["ELSE" *Substitution*]
"END"

Substitution_sélection ::=
"SELECT" *Prédicat* "THEN" *Substitution*
("WHEN" *Prédicat* "THEN" *Substitution*)*
["ELSE" *Substitution*]
"END"

Substitution_cas ::=
"CASE" *Expression* "OF"
"EITHER" *Terme_simple*$^{+","}$ "THEN" *Substitution*
("OR" *Terme_simple*$^{+","}$ "THEN" *Substitution*)*
["ELSE" *Substitution*]
"END"
"END"

Substitution_choix_non_borné ::=

"ANY" Ident$^{+","}$ "WHERE" *Prédicat* "THEN" *Substitution* "END"

Substitution_définition_locale ::=
"LET" Ident$^{+","}$ "BE"
(Ident "=" *Expression*)$^{+"\wedge"}$
"IN" *Substitution* "END"

Substitution_devient_elt_de ::=

Ident_ren$^{+","}$ ":∈" *Expression*

Substitution_devient_tel_que ::=

Ident_ren$^{+","}$ ":" "(" *Prédicat* ")"

Substitution_variable_locale ::=

"VAR" Ident$^{+","}$ "IN" *Substitution* "END"

Substitution_séquence ::=

Substitution ";" *Substitution*

Substitution_appel_opération ::=
[*Ident_ren*$^{+","}$ "⟵"] *Ident_ren* ["(" *Expression*$^{+","}$ ")"]

Substitution_simultanée ::=
Substitution "||" *Substitution*

Substitution_corps_opération ::=
Substitution_bloc
| *Substitution_identité*
| *Substitution_devient_égal*
| *Substitution_précondition*
| *Substitution_assertion*
| *Substitution_choix_borné*
| *Substitution_conditionnelle*
| *Substitution_sélection*
| *Substitution_cas*
| *Substitution_any*
| *Substitution_let*
| *Substitution_devient_elt_de*
| *Substitution_devient_tel_que*
| *Substitution_variable_locale*
| *Substitution_appel_opération*

Instruction_corps_opération ::=
Instruction_bloc
| *Instruction_variable_locale*
| *Substitution_identité*
| *Instruction_devient_égal*
| *Instruction_appel_opération*
| *Instruction_conditionnelle*
| *Instruction_cas*
| *Instruction_assertion*
| *Substitution_tant_que*

B.1.9 Règles de syntaxe utiles

Liste_ident ::= Ident
| "(" Ident$^{+","}$ ")"

Ident_ren ::= Ident$^{+"."}$

B.2 Grammaire des prédicats de typage

Typage_donnée_abstraite ::=
Ident$^{+","}$ "∈" *Expression*$^{+"×"}$
| Ident "⊆" *Expression*
| Ident "⊂" *Expression*
| Ident$^{+","}$ "=" *Expression*$^{+","}$

Typage_cte_concrète ::=
Ident$^{+","}$ "∈" *Typage_appartenance_donnée_concrète*$^{+"×"}$
| Ident "=" *Typage_égalité_cte_concrète*
| Ident "⊆" *Ensemble_simple*
| Ident "⊂" *Ensemble_simple*

Typage_appartenance_donnée_concrète ::=
Ensemble_simple
| *Ensemble_simple*$^{+"×"}$ "→" *Ensemble_simple*
| *Ensemble_simple*$^{+"×"}$ "↣" *Ensemble_simple*
| *Ensemble_simple*$^{+"×"}$ "↠" *Ensemble_simple*
| *Ensemble_simple*$^{+"×"}$ "⤖" *Ensemble_simple*
| "{" *Terme_simple*$^{+","}$ "}"
| "struct" "(" (Ident ":" *Typage_appartenance_donnée_concrète*)$^{+","}$ ")"

Typage_égalité_cte_concrète ::=
Terme
| *Expr_tableau*
| *Intervalle*
| *Ensemble_nombres_B0*
| "rec" "(" ([Ident ":"] *Terme*)$^{+","}$ ")"

Ensemble_simple ::=
Ensemble_nombres_B0
| "BOOL"
| *Intervalle_B0*
| Ident

Ensemble_nombres_B0 ::=
"NAT"
| "NAT_1"
| "INT"

Expr_tableau ::=
Ident
| "{" (*Terme_simple*$^{+"\mapsto"}$ "$\mapsto$" *Terme*)$^{+","}$ "}"
| *Ensemble_simple*$^{+"\times"}$ "$\times$" "{" *Terme* "}"

Intervalle_B0 ::=
Expression_arithmétique ".." *Expression_arithmétique*
| *Ensemble_nombres_B0*

Typage_var_concrète ::=
Ident$^{+","}$ "$\in$" *Typage_appartenance_donnée_concrète*$^{+"\times"}$
| Ident "=" *Terme*

Typage_param_entrée ::=
Ident$^{+","}$ "$\in$" *Typage_appartenance_param_entrée*$^{+"\times"}$
| Ident "=" *Terme*

Typage_appartenance_param_entrée ::=
Ensemble_simple
| *Ensemble_simple*$^{+"\times"}$ "$\rightarrow$" *Ensemble_simple*
| *Ensemble_simple*$^{+"\times"}$ "$\rightarrowtail$" *Ensemble_simple*
| *Ensemble_simple*$^{+"\times"}$ "$\twoheadrightarrow$" *Ensemble_simple*
| *Ensemble_simple*$^{+"\times"}$ "$\rightarrowtail\!\!\!\!\twoheadrightarrow$" *Ensemble_simple*
| "{" *Terme_simple*$^{+","}$ "}"
| "struct" "(" (Ident ":" *Typage_appartenance_donnée_concrète*)$^{+","}$ ")"
| "STRING"

Typage_param_mch ::=
Ident$^{+","}$ "$\in$" *Typage_appartient_param_mch*$^{+"\times"}$
| Ident$^{+","}$ "=" *Terme*$^{+","}$

Typage_appartient_param_mch ::=
Ensemble_nombres
| "BOOL"
| *Intervalle_B0*
| Ident

Ensemble_nombres ::=
"$\mathbb{Z}$"
| "$\mathbb{N}$"
| "$\mathbb{N}_1$"
| "NAT"
| "NAT_1"
| "INT"

B.3 Grammaire des types B

Type ::= *Type_de_base*
| "ℙ" "(" *Type* ")"
| *Type* "×" *Type*
| "struct" "(" (Ident ":" *Type*)$^{+","}$ ")"
| "(" *Type* ")"

Type_de_base ::=
"ℤ"
| "BOOL"
| "STRING"
| Ident

ANNEXE C TABLES DE VISIBILITE

Les règles de visibilité entre un composant *C1* et un composant *C2* définissent pour chaque constituant de *C2*, les modes d'accès applicables dans les clauses de *C1*. Pour des données, on distingue l'accès en lecture seule, en lecture et écriture ou en écriture seule. Pour des opérations, on distingue l'accès aux opérations de consultation (les opérations dont la spécification ne modifie pas les variables de la machine) et aux opérations de modification.

Dans les tables de visibilité ci-dessous M_A désigne une machine abstraite ou un raffinement, M_{N-1} désigne un raffinement ou une machine abstraite, M_N désigne un raffinement ou une implantation et M_B désigne une machine abstraite reliée à un composant par une clause de visibilité IMPORTS, SEES, INCLUDES ou USES.

Le tableau ci-dessous indique les différents modes de visibilité des constituants dans des clauses :

Mode de visibilité	Description
	constituant non visible
visible	constituant visible
visible - modifiable	constituant visible, si le constituant est une variable utilisée dans une substitution, la variable est modifiable, si le constituant est une opération appelée dans une substitution, l'opération peut modifier les variables de sa machine abstraite
visible - non modifiable	constituant visible, si le constituant est une variable utilisée dans une substitution, la variable n'est pas modifiable, si le constituant est une opération appelée dans une substitution, c'est une opération qui ne modifie pas les variables de sa machine abstraite

C.1 Visibilité dans une machine abstraite M_A

Clauses de M_A / Constituants de M_A	CONSTRAINTS	Paramètres d'INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS
Paramètres formels	visible	visible		visible	visible
Ensembles, énumérés littéraux, constantes concrètes		visible	visible	visible	visible
Constantes abstraites, non homonymes		visible	visible	visible	visible
Variables concrètes, non homonymes				visible	visible - modifiable
Variables abstraites, non homonymes				visible	visible - modifiable
Opérations propres (non promues)					

C.2 Visibilité d'une machine *vue* M_B par une machine ou un raffinement M_A

Clauses de M_A / Constituants de M_B	CONSTRAINTS	Paramètres d'INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS
Paramètres formels					
Ensembles, énumérés littéraux, constantes concrètes		visible	visible	visible	visible
Constantes abstraites		visible	visible	visible	visible
Variables concrètes					visible – non modifiable
Variables abstraites					visible – non modifiable
Opérations					visible – non modifiable

C.3 Visibilité d'une machine *incluse* M_B par une machine ou un raffinement M_A

Clauses de M_A / Constituants de M_B	CONSTRAINTS	Paramètres d'INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS
Paramètres formels					
Ensembles, énumérés littéraux, constantes concrètes			visible	visible	visible
Constantes abstraites			visible	visible	visible
Variables concrètes				visible	visible – non modifiable
Variables abstraites				visible	visible – non modifiable
Opérations					visible modifiable

C.4 Visibilité d'une machine *utilisée* (USES) M_B par une machine M_A

Clauses de M_A / Constituants de M_B	CONSTRAINTS	Paramètres d'INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS
Paramètres formels				visible	visible
Ensembles, énumérés littéraux, constantes concrètes			visible	visible	visible
Constantes abstraites			visible	visible	visible
Variables concrètes				visible	visible – non modifiable
Variables abstraites				visible	visible – non modifiable
Opérations					

C.5 Visibilité dans un raffinement M_N

Clauses de M_N / Constituants de M_N	Paramètres d'INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS
Paramètres formels	visible		visible	visible
Ensembles, énumérés littéraux, constantes concrètes, non homonymes	visible	visible	visible	visible
Constantes abstraites non homonymes	visible	visible	visible	visible
Variables concrètes non homonymes			visible	visible - modifiable
Variables abstraites non homonymes			visible	visible - modifiable
Opérations propres (non promues)				

C.6 Visibilité dans un raffinement M_N par rapport à son abstraction M_{N-1}

Clauses de M_N / Constituants de M_{N-1}	Paramètres d'INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS	
				Substitutions	Prédicats d'ASSERT
Constantes abstraites disparaissant dans M_N		visible	visible		visible
Variables abstraites disparaissant dans M_N			visible		visible

C.7 Visibilité dans une implantation M_N

Clauses de M_N / Constituants de M_N	Paramètres d'IMPORTS / EXTENDS	PROPERTIES	VALUES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS		LOCAL_ OPERATIONS
					Instructions	variants et invariants de boucles, prédicats d'ASSERT	
Paramètres formels	visible			visible	visible	visible	visible
Ensembles énumérés, énumérés littéraux, non homonymes	visible	visible	visible	visible	visible	visible	visible
Ensembles abstraits, constantes concrètes, non homonymes	visible	visible	visible modifiable	visible	visible	visible	visible
Variables concrètes non homonymes				visible	visible – modifiable	visible	visible - modifiable
Opérations propres (non promues)							
Opérations locales					visible – modifiable dans OPERATIONS, pas dans INITIALISATION		

C.8 Visibilité dans une implantation M_N par rapport à son abstraction M_{N-1}

Clauses de M_N / Constituants de M_{N-1}	Paramètres d'IMPORTS / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS		LOCAL_OPERATIONS	
				Instructions	Variants et invariants de boucles, prédicats d'ASSERT	Substitutions	Prédicats d'ASSERT
Constantes abstraites		visible	visible		visible		visible
Variables abstraites			visible		visible		visible

C.9 Visibilité d'une machine *vue* M_B par une implantation M_N

Clauses de M_N / Constituants de M_B	Paramètres d'IMPORTS / EXTENDS	PROPERTIES	VALUES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS		LOCAL_ OPERATIONS
					Instructions	Variants et invariants de boucles, prédicats d'ASSERT	
Paramètres formels							
Ensembles, énumérés littéraux, constantes concrètes	visible	visible	visible	visible	visible	visible	visible
Constantes abstraites		visible		visible		visible	visible
Variables concrètes					visible – non modifiable	visible	visible – non modifiable
Variables abstraites						visible	visible – non modifiable
Opérations					visible – non modifiable		visible – non modifiable

C.10 Visibilité d'une machine *importée* M_B par une implantation M_N

Clauses de M_N / Constituants de M_B	Paramètres d'IMPORTS / EXTENDS	PROPERTIES	VALUES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS		LOCAL_ OPERATIONS
					Instructions	Variants et invariants de boucles, prédicats d'ASSERT	
Paramètres formels							
Ensembles, énumérés littéraux, constantes concrètes		visible	visible	visible	visible	visible	visible
Constantes abstraites		visible		visible		visible	visible
Variables concrètes				visible	visible – non modifiable	visible	visible - modifiable
Variables abstraites				visible		visible	visible - modifiable
Opérations					visible – modifiable		visible - modifiable

ANNEXE D GLOSSAIRE

Abstraction

notion symétrique du raffinement. Si le composant M_n est un raffinement du composant M_{n-1}, alors M_{n-1} est une abstraction de M_n.

B0

partie du langage B qui sert directement à produire un programme informatique à partir d'un module B. Le B0 est constitué de certains constituants du module et du corps associé à ces constituants. Les constituants B0 d'un module sont déclarés dans la machine abstraite du module (la machine abstraite, ses paramètres, ses opérations), dans ses raffinements (les ensembles abstraits et énumérés, les éléments énumérés, les constantes concrètes et les variables concrètes) ou dans son implantation (liens IMPORTS et SEES de l'implantation). Les corps des constituants sont uniquement présents dans l'implantation. Ils sont situés dans les clauses IMPORTS, VALUES, INITIALISATION, PROMOTES et OPERATIONS. Les prédicats, expressions et substitutions B0, sont appelés respectivement des conditions, des termes et des instructions.

Clause

Les composants sont constitués de clauses. Chaque clause permet de déclarer une partie spécifique du composant.

Clause de visibilité

clause appartenant à l'ensemble des clauses d'un composant qui déclarent les liens entre ce composant et des instances de machines. Les clauses de visibilité sont au nombre de cinq : IMPORTS, SEES, INCLUDES, USES et EXTENDS.

Composant

désigne indifféremment une machine, un raffinement ou une implantation.

Condition

prédicat implémentable utilisé dans les instructions IF et WHILE.

Constante

désigne indifféremment une constante concrète ou une constante abstraite.

Constante abstraite

donnée de valeur constante, dont le type est quelconque, appartenant à un composant et qui pourra être raffinée au cours du raffinement du composant.

Constante concrète

donnée de valeur constante appartenant à un composant qui représente soit un scalaire, soit un tableau, soit un intervalle fini d'entiers ou d'éléments d'ensemble abstrait. Une constante concrète est automatiquement conservée au cours du raffinement.

Constituant

Un constituant désigne tout ce qui peut être nommé dans un composant. Il peut s'agir d'un ensemble, d'une constante, d'une variable, d'une variable muette, d'une variable locale, d'un paramètre de machine, d'un paramètre d'opération ou d'une opération.

Démonstration

cf. Preuve

Donnée

objet mathématique possédant un nom et une valeur. Le type d'une donnée B doit correspondre aux types définis dans la bibliothèque mathématique.

Implantation

dernier raffinement d'un module développé. Une implantation est constituée principalement de B0.

Initialisation

L'initialisation d'une instance d'un composant est décrite dans la clause INITIALISATION. Elle permet notamment de donner une valeur initiale aux variables de l'instance du composant.

Instance de machine

copie dont le modèle est une machine abstraite. Une instance de machine abstraite possède un espace de données qui contient les valeurs des données modifiables de la machine (les variables et les paramètres de machine).

Instance de machine abstraite

instance créée en phase de spécification par *inclusion*. Elle constitue un espace de données abstrait.

Instance de machine concrète

instance créée en phase d'implantation par *importation*. Elle constitue un espace de données concret du programme informatique associé au projet.

Instance de machine *importée*

instance de machine figurant dans la clause IMPORTS ou EXTENDS.

Instance de machine *incluse*

instance de machine figurant dans la clause INCLUDES ou EXTENDS.

Instance de machine *utilisée*

instance de machine figurant dans la clause USES d'une machine.

Instance de machine *vue*

instance de machine figurant dans la clause SEES.

Instruction

substitution faisant partie du B0.

Invariant

prédicat exprimant des propriétés portant sur les données d'un composant. On distingue deux sortes d'invariants dans le langage B : l'invariant de la clause INVARIANT, qui porte sur les données du composant et l'invariant de boucle WHILE qui porte sur les données utilisées dans une boucle « tant que ».

Invariant de liaison

invariant particulier de la clause INVARIANT d'un composant, qui exprime une relation de raffinement entre les variables du composant et les variables de son abstraction.

Lexème

chaîne de caractères qui appartient à une unité lexicale d'un langage. Le résultat de la phase d'analyse lexicale d'un texte est une suite de lexèmes.

Machine

cf. Machine abstraite

Machine abstraite

spécification d'un module B. Une machine abstraite est constituée de clauses qui permettent de déclarer les liens de la machine abstraite, sa partie statique (ensembles, paramètres, constantes, variables et leurs propriétés) et sa partie dynamique (initialisation des variables et opérations sur les données).

Machine de base

cf. Module de base

Machine principale

machine particulière d'un projet qui sert de point d'entrée pour l'exécution du code d'un projet B.

Machine requise

machine qui est *vue*, *incluse*, *utilisée*, *importée* ou raffinée.

Module

Un module B permet de modéliser un sous-système ; il constitue une partie d'un projet B. La spécification d'un module est formalisée en langage B dans une machine abstraite. Il existe trois sortes de modules, les modules développés par raffinements successifs d'une machine abstraite, les modules abstraits et les modules de base. Par abus de langage, on confond souvent le module et sa spécification, la machine abstraite.

Module abstrait

désigne un module B composé d'une machine abstraite qui n'est pas raffinée et qui ne possède pas de code associé. Un module abstrait sert principalement à être *inclus* dans une machine ou un raffinement.

Module de base

désigne un module B composé d'une machine abstraite qui n'est pas raffinée et qui possède un code associé manuellement, implémentant directement les données et les services de la machine.

Module développé

Un module développé est un module entièrement développé en langage B. Il est constitué d'une machine abstraite, de ses éventuels raffinements et de son implantation.

Non liberté

Une variable est dite non libre dans un prédicat ou dans une expression si elle n'est pas présente dans la formule ou si elle est présente dans des sous-formules qui sont sous la portée de certains quantificateurs introduisant une variable quantifiée de même nom.

Obligation de preuve

lemme mathématique constitué d'une liste de prédicats appelés hypothèses et d'un prédicat appelé but qui doit être prouvé sous ces hypothèses.

Opération

service offert par un module B. Les opérations constituent la partie dynamique d'un module. Par défaut, lorsque l'on parle d'opération sans ajouter de qualificatif, il s'agit d'une opération non locale, c'est-à-dire d'une opération définie dans un module B, utilisable par les utilisateurs du module.

Opération de consultation

opération dont la spécification dans une machine abstraite ne modifie pas les variables de la machine.

Opération locale

opération locale à une implantation : elle est spécifiée et implémentée dans une implantation et utilisable seulement par cette implantation. Les opérations locales sont spécifiées dans la clause LOCAL_OPERATIONS et sont implantées dans la clause OPERATIONS.

Opération promue

opération d'un composant dont les paramètres et la spécification sont identiques à une opération d'une instance de machine *incluse* ou *importée*.

Opération propre

opération (non locale) d'un composant dont le corps est donné dans le composant, au sein de la clause OPERATIONS.

Paramètre

En B, il est possible de paramétrer les machines abstraites, les définitions et les opérations. Les paramètres formels sont des noms donnés lors de la déclaration d'un constituant paramétré à ses paramètres. Lors de l'utilisation d'un constituant, on attribue à chaque paramètre une valeur appelée paramètre effectif.

Preuve

activité mathématique consistant à démontrer la véracité d'Obligations de Preuve. Le développement d'un projet comporte principalement deux grandes activités : l'écriture de composants et la preuve des Obligations de Preuves associées à ces composants.

Projet

désigne un ensemble complet et autosuffisant de modules permettant de spécifier de manière formelle un système et éventuellement d'engendrer un programme informatique conforme aux spécifications formelles.

Raffinement

Le raffinement noté M_n d'un composant noté M_{n-1} est une nouvelle formulation de M_{n-1}, dans laquelle certains constituants de M_{n-1} sont raffinés (les constantes abstraites et les variables abstraites, l'initialisation, les opérations).

Raffiner

Raffiner un constituant possédant certaines propriétés, c'est offrir une nouvelle formulation de ce constituant à l'aide d'un ou de plusieurs nouveaux constituants qui ne doivent pas contredire les propriétés du constituant raffiné et diminuer le niveau d'abstraction et d'indéterminisme du constituant. Raffiner permet également d'enrichir un composant par rapport à sa spécification.

Renommage

Le renommage en B permet de créer des instances de machines abstraites. Une instance de machine est désignée par le nom de la machine précédé d'un préfixe de renommage. Le préfixe de renommage est constitué d'un identificateur suivi d'un point. Les variables, les opérations et les paramètres d'une instance de machine renommée sont désignés de l'extérieur à l'aide du même préfixe de renommage.

Signature d'une opération

liste ordonnée des types des paramètres d'entrée et de sortie d'une opération.

Substitution

notation mathématique permettant de modéliser la transformation de formules mathématiques.

Tableau

fonction totale d'un ensemble simple, ou d'un produit cartésien d'ensembles simples (si le tableau est multidimensionnel), vers un ensemble simple.

Typage

mécanisme de vérification statique des données. Le type d'une donnée d est le plus grand ensemble (parmi les ensembles définis dans le langage B) auquel appartient d.

Valuation

mécanisme qui consiste à donner des valeurs aux constantes concrètes et aux ensembles abstraits déclarés dans un module B, au sein de l'implantation du module. La valuation est décrite dans la clause VALUES.

Variable

désigne indifféremment une variable concrète ou une variable abstraite.

Variable abstraite

donnée appartenant à un composant dont le type est quelconque et qui est raffinée au cours du raffinement du composant.

Variable concrète

donnée appartenant à un composant et conservée au cours du raffinement qui représente soit un scalaire, soit un tableau.

ANNEXE E INDEX

B

C

D

E