Quelques notions de type en informatique

Dominique Duval,
LMC-IMAG-UJF, Grenoble
Dominique Duval@imag.fr

Cours de 3ème année, IUP MAI, 2003-04

Déduire et démontrer
sont des opérations intellectuelles rigoureuses et enthousiasmantes,
mais encore faut-il que la matière ne soit pas seulement des mots.

Jacques PAUL, Histoire intellectuelle de l'Occident médiéval.

Introduction (21/10/03)

Historiquement, la notion de *type* est à l'interface de la logique, des mathématiques et de l'informatique.

La *théorie des types* a été introduite par RUSSELL en 1903, afin d'éviter les paradoxes de la théorie des *ensembles*, elle-même introduite par CANTOR en 1874.

En informatique, les premières apparitions de *systèmes de types* servaient à distinguer les représentations en machine des nombres entiers et des nombres flottants, en particulier pour le langage Fortran, conçu par BACKUS en 1954.

De façon générale, les *types* servent à structurer les données.

Par exemple, considérons les nombres entiers.
– Du point de vue mathématique, les entiers forment un *ensemble*, qui est généralement noté $\mathbb{Z}$. Cet ensemble est défini à l'aide de formules logiques, par un procédé complexe.
– Cependant, on fait couramment des calculs sur les entiers sans connaître la définition précise de $\mathbb{Z}$. Il suffit pour cela de connaître quelques entiers, de savoir quelles opérations sont permises sur les entiers, combien elles ont d'arguments, et comment elles fonctionnent.
– En informatique, le *type* entier est formé d'un *nom*, par exemple `int`, auquel sont associés quelques *symboles de constantes* comme 0 et 1, et quelques *symboles d'opération* comme $+$ et $-$ avec leur *arité*, c'est-à-dire le nombre de leurs arguments (les deux opérations $+$ et $-$ sont binaires).
– Si on considère les types d'un point de vue *abstrait*, ce qui est le cas dans ce cours, alors la *représentation* des entiers en machine (comme suite de bits par exemple), et l'*implantation* des opérations, ne font pas partie du type entier. Ainsi *un unique type peut avoir plusieurs implantations différentes*.
– Entre ces deux extrêmes (seulement les noms, ou toute l'implantation), on peut se demander si les *propriétés* des opérations doivent faire partie du type. Par exemple, le fait que l'addition est commutative (c'est-à-dire $x+y = y+x$ pour tous entiers x et y) doit-il faire partie du type entier ? Dans ce cours, on verra des situations où la réponse est *oui*, comme pour les types abstraits de données, et des situations où la réponse est *non*, comme pour les langages orientés objet. En effet, même s'il semble plus satisfaisant que le type tienne compte des propriétés des opérations, celles-ci sont souvent difficiles, voire impossibles, à vérifier...Cependant, des réponses intermédiaires, du style *oui si on veut*, existent aussi ; c'est le cas de JML (Java Modeling Langage) : c'est un langage de spécification pour Java, qui permet de préciser certaines propriétés des opérations, et de les vérifier lorsque c'est possible.

En résumé, on définit les données en deux phases :
1. *spécification* (partie abstraite) : on nomme les opérations et on précise certaines de leurs propriétés : au moins le type des arguments et du résultat, et éventuellement d'autres propriétés ;
2. *implantation* (partie concrète) : on choisit une représentation pour ces données, et on écrit le corps des opérations.
Le typage, au sens de ce cours, ne concerne que la partie abstraite.

Un système de types pour un langage de programmation doit permettre d'assurer, au moins, les propriétés suivantes :

- *Sûreté.* La vérification des types des programmes doit éviter (soit à la compilation soit à

l'exécution) l'exécution de certaines opérations illégales. Par exemple, ajouter une chaîne à un entier est une erreur de typage, qui doit être repérée.
Sûreté ou *safety. [www.spacenow.org.uk, 9 Sep 2003] "Smarter, stronger, safer" is NASA's new mantra replacing the "faster, better, cheaper" guidelines the agency adhered to throughout the 1990s.*

- *Optimisation.* Le typage fournit des informations au compilateur ou à l'interpréteur, en particulier pour réserver de la place en mémoire ou pour sélectionner le code à exécuter.

- *Documentation.* Le typage fournit des informations au programmeur qui peuvent être vues comme une ébauche de documentation sur la façon dont les opérations peuvent, et doivent, être utilisées.

- *Abstraction.* Le fait de nommer des types, et surtout de distinguer les types de leur implantation, oblige le programmeur à penser à un certain niveau d'abstraction. C'est une démarche importante pour la modélisation. De plus, la correction d'un programme doit dépendre uniquement de la spécification des opérations et pas de la représentation des données ni de l'implantation des opérations, qui peuvent être modifiées.

Par ailleurs, les types sont interdépendants, car les opérations font souvent intervenir plusieurs types. Par exemple élever un réel à une puissance entière fait intervenir le type réel et le type entier. Une famille de types et d'opérations, avec éventuellement une description de certaines propriétés de ces opérations, est appelée une *spécification*, elle détermine la *syntaxe*, c'est-à-dire la forme, des chaînes de caractères qu'on peut utiliser.

La signification qu'on donne à cette spécification forme sa *sémantique* : la sémantique du nom `int` est fournie par l'ensemble $\mathbb{Z}$. On dit aussi que $\mathbb{Z}$ forme un *modèle* du type `int`.

machine	SYNTAXE	spécification
humain	SEMANTIQUE	modèle(s)

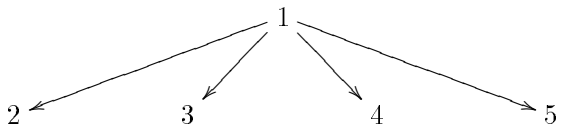
Par ailleurs, les *programmes* associés à une spécification sont formés à partir des opérations, en suivant certaines *règles*, par exemple "`si` $(3-4+2)>0$ `alors` $5+5$ `sinon` 100" est un programme sur les entiers, de valeur 10 puisque $(3-4+2)=1>0$ et $5+5=10$.

Le but du cours est de fournir quelques clés pour comprendre les notions de type et de spécification en informatique. Ces notions sont présentées à partir d'exemples et de diagrammes, de façon évidemment très incomplète. Le cours est descriptif : il comprend peu d'énoncés de théorèmes, et aucune preuve.

Dans tout le cours, de façon parfois caricaturale, on tente de bien distinguer deux niveaux : d'une part le niveau de la machine (la syntaxe) et d'autre part le niveau humain (la sémantique). Les ensembles servent à exprimer la sémantique, et les types servent à représenter les ensembles dans la syntaxe.

Plan du cours :
1. Fonctions
2. Si...alors...sinon...
3. Récurrence et types abstraits de données
4. Types de fonctions et lambda-calcul typé
5. Types et classes

Le chapitre 1 introduit les bases du formalisme indispensable à la compréhension des chapitres suivants, qui sont indépendants les uns des autres :



Il existe de très nombreux articles et ouvrages sur les types abstraits de données et le lambda-calcul typé, permettant d'aller bien au-delà de la présentation succinte qui est faite dans ce cours. Ces notions sont aussi présentées dans le livre suivant, qui étudie en détail le typage dans les langages objets :
K.B. Bruce. *Foundations of Object-Oriented Languages. Types and Semantics.* Massachussets Institute of Technology (2002).

1 Fonctions (29/09/03)

Un *type* en informatique correspond à un *ensemble* en mathématiques; de façon similaire, une *fonction* en informatique correspond à une *application* en mathématiques.

Cependant, comme le mot *fonction* peut avoir plusieurs significations, commençons par préciser la terminologie choisie dans ce cours : on utilise le mot *fonction* seulement dans son cadre informatique, et dans son sens le plus restreint.

En mathématiques, une *application* φ d'un ensemble A vers un ensemble B associe à tout élément a de A un élément $b = f(a)$ de B. Traditionnellement, on note :

$$\varphi : A \to B : a \mapsto \varphi(a) \ .$$

Par exemple :

$$\mathit{double} : \mathbb{N} \to \mathbb{N} : x \mapsto 2x \ .$$

Souvent, mais pas dans ce cours, une application s'appelle aussi une fonction, ou une fonction totale.

En informatique, le sens du mot *fonction* dépend des langages...Ici on distingue, d'une part les *fonctions*, dont la valeur ne dépend pas de l'état courant, et dont l'exécution ne modifie pas l'état, et d'autre part les *procédures*, dont la valeur peut dépendre de l'état, et qui peuvent modifier l'état. Dans ce chapitre on étudie seulement les fonctions, plus tard on considèrera les procédures. Dans le style du langage C, si f désigne une fonction avec un argument de type X et qui retourne un résultat de type Y, on note :

$$\texttt{Y f(X x)} \ \{\ldots\} \ .$$

Par exemple :

$$\texttt{Int double(Int x)} \ \{\texttt{return(x + 2);}\} \ .$$

En résumé, il faut bien distinguer deux niveaux :

informatique	machine	SYNTAXE	type	fonction
mathématiques	humain	SEMANTIQUE	ensemble	application

1.1 Fonctions à un argument

Considérons l'*ensemble* :

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \ .$$

C'est l'ensemble des entiers des mathématiciens, il est infini. On va considérer pour le moment deux opérations sur l'ensemble $\mathbb{Z}$, qui ont chacune un argument : l'opération successeur s et l'opération prédécesseur p. L'opération s est une application qui à tout entier associe un entier, et de même pour p :

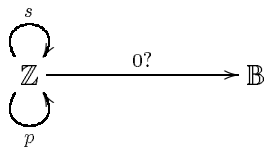
$$s : \mathbb{Z} \to \mathbb{Z} : x \mapsto s(x) = x + 1 \ ,$$

$$p : \mathbb{Z} \to \mathbb{Z} : x \mapsto p(x) = x - 1 \ .$$

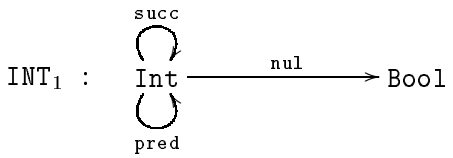
On considère aussi le test d'égalité à 0 ; en notant $\mathbb{B} = \{V, F\}$ l'ensemble des booléens, le test d'égalité à 0 est une application :

$$0? : \mathbb{Z} \to \mathbb{B} : x \mapsto (x = 0) \ .$$

On peut grouper ces deux ensembles et ces trois applications dans un seul graphe (dans ce cours, tous les graphes sont orientés) :



On peut aussi reprendre cet exemple dans l'autre sens : d'abord, on dessine un graphe *abstrait* $\texttt{INT}_1$:

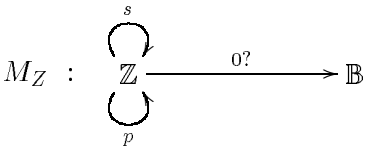


Ce graphe correspond à une *spécification* dans un langage de programmation, ses points sont des *types* et ses flèches sont des *fonctions* :

Spécification $\texttt{INT}_1$:
Types : `Int`, `Bool`;
Fonctions : `succ` : `Int` $\to$ `Int`, `pred` : `Int` $\to$ `Int`, `nul` : `Int` $\to$ `Bool`.

Ensuite, on dit qu'un *modèle* de cette spécification, c'est un graphe *concret* de la même forme, c'est-à-dire que chaque type est remplacé par un *ensemble* et chaque fonction par une *application*.

En particulier, le graphe concret qu'on a construit à partir des entiers est un modèle de $\texttt{INT}_1$, on le note M_Z :



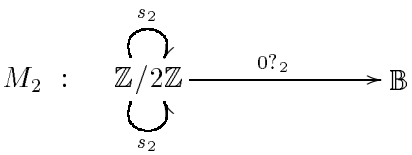
On dit que ce modèle M_Z *interprète* le type `Int` par l'ensemble $\mathbb{Z}$, le type `Bool` par l'ensemble $\mathbb{B}$, qu'il *interprète* la fonction `succ` par la fonction s, la fonction `pred` par la fonction p, et la fonction `nul` par la fonction 0?. On note cela :

$$M_Z(\texttt{Int}) = \mathbb{Z}\ ,\ M_Z(\texttt{Bool}) = \mathbb{B}\ ,\ M_Z(\texttt{succ}) = s\ , M_Z(\texttt{pred}) = p\ , M_Z(\texttt{nul}) = 0?\ .$$

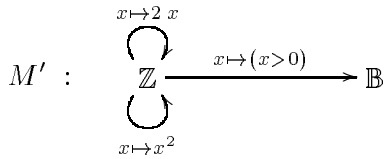
Mais $\texttt{INT}_1$ a beaucoup d'autres modèles. Par exemple, notons $\mathbb{Z}/2\mathbb{Z} = \{0, 1\}$, notons $s_2 : \mathbb{Z}/2\mathbb{Z} \to \mathbb{Z}/2\mathbb{Z}$ l'application définie par $s_2(0) = 1, s_2(1) = 0$, et $0?_2 : \mathbb{Z}/2\mathbb{Z} \to \mathbb{B}$ l'application définie par $0?_2(0) = V, 0?_2(1) = F$. Alors $\texttt{INT}_1$ a pour modèle "des entiers modulo 2" le modèle M_2 défini par :

$$M_2(\texttt{Int}) = \mathbb{Z}/2\mathbb{Z}\ ,\ M_2(\texttt{Bool}) = \mathbb{B}\ ,\ M_2(\texttt{succ}) = s_2\ , M_2(\texttt{pred}) = s_2\ , M_2(\texttt{nul}) = 0?_2\ .$$

Autrement dit :



L'interprétation des types `Int` et `Bool` ne suffisent pas à déterminer le modèle. Par exemple, voici un autre modèle M' de $\mathtt{INT}_1$, tel que $M'(\mathtt{Int}) = \mathbb{Z}$ et $M'(\mathtt{Bool}) = \mathbb{B}$, comme pour M_Z, mais qui est différent de M_Z :

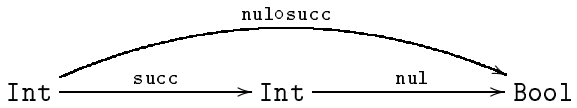


SYNTAXE	type	fonction	spécification	$\mathtt{INT}_1$
SEMANTIQUE	ensemble	application	modèle	M_Z, M_2, M',...

On peut ajouter des fonctions à la spécification $\mathtt{INT}_1$ en construisant des chemins à partir des fonctions déjà connues.

Etant donnée une spécification S, toutes les fonctions, celles qui sont données dès le départ comme celles qui peuvent être ajoutées par la suite, sont appelées les *fonctions engendrées par* S ou les *termes* de S.

Par exemple, le chemin formé de $\mathtt{succ} : \mathtt{Int} \to \mathtt{Int}$ suivi de $\mathtt{nul} : \mathtt{Int} \to \mathtt{Bool}$ est la fonction notée $\mathtt{nul} \circ \mathtt{succ} : \mathtt{Int} \to \mathtt{Bool}$:



Remarque : Pour y voir plus clair, comme ci-dessus, on a le droit de dessiner plusieurs copies d'un seul type ou d'une seule fonction, à condition de bien indiquer son nom.

Si M est un modèle de $\mathtt{INT}_1$, on définit l'interprétation de la fonction $\mathtt{nul} \circ \mathtt{succ}$ dans M par :

$$M(\mathtt{nul} \circ \mathtt{succ}) = M(\mathtt{nul}) \circ M(\mathtt{succ}) \ ,$$

où $M(\mathtt{nul}) \circ M(\mathtt{succ})$ désigne, comme d'habitude, l'application *composée* de $M(\mathtt{succ})$ et $M(\mathtt{nul})$:

$$M(\mathtt{nul}) \circ M(\mathtt{succ}) : x \mapsto M(\mathtt{nul})(M(\mathtt{succ})(x)) \ .$$

Par exemple, dans le modèle M_Z, on interprète la fonction $\mathtt{nul} \circ \mathtt{succ}$ par l'application :

$$0? \circ s : \mathbb{Z} \to \mathbb{B} : x \mapsto 0?(s(x)) \ ,$$

qui vaut V si $x = -1$ et F sinon. Voici quelques fonctions de $\mathtt{INT}_1$:

$$\mathtt{succ} : \mathtt{Int} \to \mathtt{Int} \ , \ \mathtt{nul} : \mathtt{Int} \to \mathtt{Bool} \ , \ \mathtt{pred} \circ \mathtt{succ} : \mathtt{Int} \to \mathtt{Int} \ , \ \mathtt{succ} \circ \mathtt{pred} : \mathtt{Int} \to \mathtt{Int} \ ,$$

$$\mathtt{pred} \circ \mathtt{succ} \circ \mathtt{succ} \circ \mathtt{pred} \circ \mathtt{pred} : \mathtt{Int} \to \mathtt{Int} \ , \mathtt{nul} \circ \mathtt{pred} \circ \mathtt{succ} : \mathtt{Int} \to \mathtt{Bool} \ .$$

Les deux fonctions $\mathtt{pred} \circ \mathtt{succ}$ et $\mathtt{succ} \circ \mathtt{pred}$ ont la même interprétation dans le modèle M_Z : le successeur du prédécesseur d'un entier x c'est x, de même que le prédécesseur du successeur de x. Cependant, les deux fonctions $\mathtt{pred} \circ \mathtt{succ}$ et $\mathtt{succ} \circ \mathtt{pred}$, au niveau de la spécification $\mathtt{INT}_1$, sont différentes. Et d'ailleurs, dans le modèle M', elles ont des interprétations différentes : $M'(\mathtt{pred} \circ \mathtt{succ}) = (2\,x)^2 = 4\,x^2$ et $M'(\mathtt{succ} \circ \mathtt{pred}) = 2(x^2) = 2\,x^2$.

On construit aussi des fonctions à partir des chemins vides, qui bien sûr ont le même type pour source et pour but. On les appelle des *fonctions identités*. Par exemple la fonction identité de `Int` est notée $\mathtt{id}_{\mathtt{Int}}$.

Si M est un modèle de $\mathtt{INT}_\mathtt{l}$, on définit l'interprétation de la fonction $\mathtt{id}_{\mathtt{Int}}$ dans M par :

$$M(\mathtt{id}_{\mathtt{Int}}) = id_{M(\mathtt{Int})} \ .$$

où $id_{M(\mathtt{Int})}$ désigne, comme d'habitude, l'*application identité* de $M(\mathtt{Int})$:

$$id_{M(\mathtt{Int})} : x \mapsto x \ .$$

Par exemple, dans le modèle M_Z, on interprète la fonction $\mathtt{id}_{\mathtt{Int}}$ par l'application :

$$id_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z} : x \mapsto x \ .$$

En conclusion, il importe de bien distinguer :
– le niveau *syntaxique*, celui de la spécification, c'est-à-dire celui de la machine, où on forme des chaînes de caractères a priori sans signification ;
– et le niveau *sémantique*, celui des modèles, c'est-à-dire celui de l'utilisateur, où on donne une signification à ces chaînes de caractères.

Par exemple, le symbole "$\circ$" de composition des fonctions n'a pas de signification au niveau de la syntaxe, il permet seulement de décrire des chemins dans un graphe ; par contre au niveau de la sémantique il signifie la composition des applications. De même, le symbole "id" permet seulement de décrire des chemins vides dans un graphe ; par contre au niveau de la sémantique il signifie les applications identités.

SYNTAXE	type	fonction	spécification
SEMANTIQUE	ensemble	application	modèle

1.2 Equations

Les applications peuvent avoir diverses propriétés, par exemple le prédécesseur du successeur d'un entier x, c'est x lui-même, autrement dit :

$$\text{pour tout } x \in \mathbb{Z} \quad p(s(x)) = x \ .$$

On peut exprimer une telle propriété au niveau de la syntaxe, grâce à une *équation*. Avec notre point de vue graphique, les variables ne sont pas faciles à traiter. Mais en considérant l'application composée :

$$p \circ s : \mathbb{Z} \to \mathbb{Z} : x \mapsto p(s(x)) \ ,$$

et l'application identité :

$$id_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z} : x \mapsto x \ ,$$

la propriété qui nous intéresse s'écrit :

$$\text{pour tout } x \in \mathbb{Z} \ , \ p \circ s(x) = id_{\mathbb{Z}}(x) \ .$$

C'est une égalité entre les deux applications $p \circ s$ et $id_{\mathbb{Z}}$, ce qui s'écrit tout simplement, sans variable :

$$p \circ s = id_{\mathbb{Z}} \ .$$

Or $p \circ s$ est l'interprétation de la fonction $\texttt{pred} \circ \texttt{succ}$ et $id_{\mathbb{Z}}$ est l'interprétation de la fonction $id_{\texttt{Int}}$. Pour traduire l'égalité $p \circ s = id_{\mathbb{Z}}$, on ajoute à $\texttt{INT}_1$ une *équation* :

$$\texttt{pred} \circ \texttt{succ} \equiv id_{\texttt{Int}} \ .$$

On utilise parfois le symbole d'égalité "=" plutôt que le symbole "≡". Mais cela peut prêter à confusion. En effet, du point de vue syntaxique, pour la machine, les fonctions $\texttt{pred} \circ \texttt{succ}$ et $id_{\texttt{Int}}$ sont deux chaînes de caractères distinctes. C'est seulement dans la sémantique, lorsqu'on choisit un modèle pour les interpréter, que ces deux foncitons deviennent des applications égales. Dans ce cours, on utilisera donc le symbole "≡".

Notons $\texttt{INT}_2$ la nouvelle spécification obtenue :

Spécification $\texttt{INT}_2$:
Types : Int, Bool;
Fonctions : $\texttt{succ} : \texttt{Int} \to \texttt{Int}$, $\texttt{pred} : \texttt{Int} \to \texttt{Int}$, $\texttt{nul} : \texttt{Int} \to \texttt{Bool}$;
Equation : $\texttt{pred} \circ \texttt{succ} \equiv id_{\texttt{Int}}$.

On a donc généralisé la notion de *spécification* : c'est un graphe **avec en plus** des équations. On généralise la notion de modèle an ajoutant que les équations (entre fonctions) doivent devenir des égalités (entre applications) : les modèles de $\texttt{INT}_2$ sont les modèles de $\texttt{INT}_1$ qui *satisfont* l'équation, c'est-à-dire tels que :

$$M(\texttt{pred} \circ \texttt{succ}) = M(\texttt{id}_{\texttt{Int}}) \ ,$$

ce qui revient à dire que :

$$M(\texttt{pred}) \circ M(\texttt{succ}) = id_{M(\texttt{Int})} \ .$$

Donc M_Z est aussi un modèle de $\texttt{INT}_2$, mais par contre M' n'est pas un modèle de $\texttt{INT}_2$.

Remarque : Les équations servent à **deux** choses. D'une part, comme on l'a vu, les équations servent à imposer des contraintes aux modèles. D'autre part, les équations servent aussi à *raisonner* et à *calculer* à partir de la spécification.

Par exemple, puisque $\texttt{pred} \circ \texttt{succ} \equiv \texttt{id}_{\texttt{Int}}$ dans la spécification $\texttt{INT}_2$, on peut en déduire que $\texttt{succ} \circ \texttt{pred} \circ \texttt{succ} \equiv \texttt{succ}$. Autrement dit, dans n'importe quel modèle M de $\texttt{INT}_2$:

$$\text{pour tout } x \in M(\texttt{Int}) \ , \ M(\texttt{succ})(M(\texttt{pred})(M(\texttt{succ})(x))) = M(\texttt{succ})(x) \ ,$$

Toutes les équations qu'on peut déduire, par des règles "de ce genre", des équations d'une spécification S, peuvent être ajoutées à S, sans modifier ses modèles. On sait ce que signifie exactement "des règles de ce genre", mais on ne va pas le définir ici, faute de temps. Toutes ces équations (celles qui sont données dès le départ comme celles qui peuvent être ajoutées par la suite) sont appelées les *équations engendrées par* S ou les *congruences* de S. Leur ensemble forme la *relation de congruence* de S.

On suppose pour le moment, comme dans l'exemple ci-dessus, que toutes les propriétés qu'on considère s'expriment avec des *équations*, c'est-à-dire des propriétés de la forme :

$$\text{pour tous } \ldots \qquad (\ldots = \ldots) \ .$$

On verra sur un exemple, dans un autre chapitre, comment on peut exprimer de propriétés plus compliquées.

SYNTAXE	type	fonction ou terme	équation ou congruence
SEMANTIQUE	ensemble	application	égalité

1.3 Fonctions à plusieurs arguments

Une idée fondamentale de cette présentation diagrammatique des fonctions, c'est qu'une fonction à deux arguments x et y est considérée comme une fonction à un seul argument, qui est le couple (x, y). Le problème est alors de bien définir les couples d'éléments.

Considérons l'opération d'addition sur les entiers. C'est une opération *binaire*, c'est-à-dire qu'elle a deux arguments. C'est donc une application qui à tout couple d'entiers associe un entier.

Or, pour tous ensembles X et Y, l'ensemble des couples (x, y) avec x dans X et y dans Y se note $X \times Y$ et s'appelle le *produit cartésien* de X et Y.

Attention ! Ici on utilise, pour une opération sur les ensembles, le même mot "produit" et le même symbole "$\times$", que pour une opération sur les nombres. Il ne faut pas les confondre, mais quand même il y a un rapport entre les deux : si les ensembles X et Y sont finis, alors l'ensemble $X \times Y$ aussi, et $Card(X \times Y) = Card(X) \times Card(Y)$.

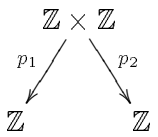
Donc l'application + peut être notée :

$$+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \ .$$

On notera parfois $+(x, y)$ (notation préfixe) au lieu de $x + y$ (notation infixe).

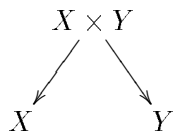
Le produit cartésien $X \times Y$ est associé à deux projections : la première projection est l'application de $X \times Y$ vers X qui envoie (x, y) sur x, et la seconde projection est l'application de $X \times Y$ vers Y qui envoie (x, y) sur y. Dans notre exemple, on note p_1 et p_2 les deux projections de $\mathbb{Z} \times \mathbb{Z}$ vers $\mathbb{Z}$:

$$p_1 : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} : (x, y) \mapsto x \ , \ p_2 : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} : (x, y) \mapsto y \ .$$



On va construire une spécification $\texttt{INT}_3$ en enrichissant $\texttt{INT}_2$. D'abord, on ajoute à $\texttt{INT}_2$ un type qu'on note I_2 et trois fonctions $\texttt{proj}_1$, $\texttt{proj}_2$ et $\texttt{ajout}$, toutes les trois de I_2 vers I. Alors on peut prolonger le modèle M_Z de $\texttt{INT}_2$ en posant $M_Z(I_2) = \mathbb{Z} \times \mathbb{Z}$, $M_Z(\texttt{proj}_1) = p_1$, $M_Z(\texttt{proj}_2) = p_2$, et $M_Z(\texttt{ajout}) = +$. Mais de plus, on veut que $M(I_2) = M(I) \times M(I)$ dans tout modèle M de $\texttt{INT}_3$, et que les fonctions $M(p_1)$ et $M(p_2)$ soient les projections.

On appelle *cône projectif (binaire)* un couple de fonctions de même source mais pas forcément de même but. Et on appelle *sommet* du cône la source commune à ses deux fonctions, et *base* du cône la liste de leurs buts. Pour dire qu'un cône projectif doit devenir un produit cartésien, on l'appelle un *produit potentiel*. On note souvent $X \times Y$ le sommet du produit potentiel de base (X, Y) :



Spécification $\texttt{INT}_3$:

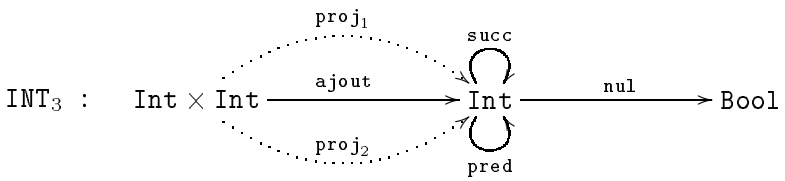
Types : $\texttt{Int}$, $\texttt{Bool}$, $\texttt{Int} \times \texttt{Int}$;

Fonctions : $\texttt{succ} : \texttt{Int} \to \texttt{Int}$, $\texttt{pred} : \texttt{Int} \to \texttt{Int}$, $\texttt{nul} : \texttt{Int} \to \texttt{Bool}$,
$\texttt{proj}_1 : \texttt{Int} \times \texttt{Int} \to \texttt{Int}$, $\texttt{proj}_2 : \texttt{Int} \times \texttt{Int} \to \texttt{Int}$, $\texttt{ajout} : \texttt{Int} \times \texttt{Int} \to \texttt{Int}$;

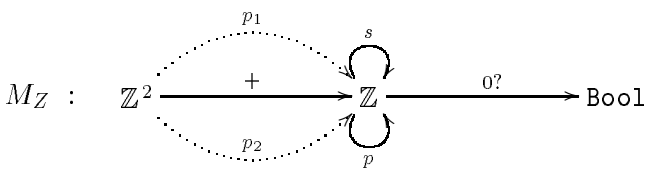
Equation : $\texttt{pred} \circ \texttt{succ} \equiv id_{\texttt{Int}}$;

Produit potentiel : (sommet $\texttt{Int} \times \texttt{Int}$, projections $\texttt{proj}_1$, $\texttt{proj}_2$);

Pour préciser que, parmi tous les cônes projectifs, certains sont des produits potentiels, on décide de représenter les projections des produits potentiels par des flèches pointillées :



M_Z se prolonge de manière unique en un modèle de $\texttt{INT}_3$ (les flèches pointillées représentent donc les projections d'un produit cartésien) :



Remarque : Les produits potentiels servent à **deux** choses. D'une part, comme on l'a vu, les produits potentiels servent à imposer des contraintes aux modèles. D'autre part, les produits potentiels servent aussi à engendrer de nouvelles fonctions et de nouvelles équations, donc à *raisonner* et *calculer* à partir de la spécification. La règle fondamentale est la *construction de paires*, qu'on va voir maintenant.

Dans la spécification $\texttt{INT}_3$, la règle de construction de paires dit que :

SI on a un type X et deux fonctions $f : X \to \texttt{Int}$ et $g : X \to \texttt{Int}$

ALORS on peut ajouter une fonction $(f, g) : X \to \texttt{Int} \times \texttt{Int}$

avec les équations $p_1 \circ (f, g) \equiv f$ et $p_2 \circ (f, g) \equiv g$.

Alors, si M est un modèle de $\texttt{INT}_2$, à cause des équations, il y a une seule façon d'interpréter la fonction (f, g) :

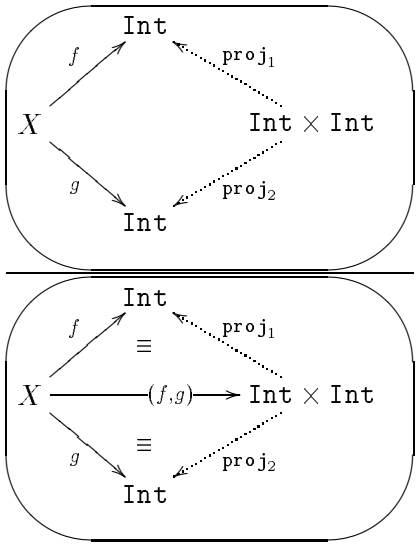
$$M((f, g)) : M(X) \to M(\texttt{Int} \times \texttt{Int}) : x \mapsto (M(f)(x), M(g)(x)) \ .$$

Par exemple, à partir de $\texttt{succ} : \texttt{Int} \to \texttt{Int}$ et $\texttt{pred} : \texttt{Int} \to \texttt{Int}$, on peut ajouter une fonction $(\texttt{succ}, \texttt{pred}) : \texttt{Int} \to \texttt{Int} \times \texttt{Int}$, dont l'unique interprétation dans M_Z est l'application $x \mapsto (s(x), p(x))$.

On a l'habitude d'écrire les règles sous forme de "fraction" $\frac{H}{C}$, avec l'hypothèse au-dessus et la conclusion en-dessous :

$$\frac{f : X \to \texttt{Int} \quad g : X \to \texttt{Int}}{(f, g) : X \to \texttt{Int} \times \texttt{Int} \quad p_1 \circ (f, g) \equiv f \quad p_2 \circ (f, g) \equiv g} \ .$$

On peut aussi dessiner cette règle :



Une telle *règle* $\frac{H}{C}$ signifie :

Toute occurrence de H dans une spécification peut être complétée par une occurrence de C, sans modifier les modèles.

Les nouvelles fonctions (f, g) permettent d'en engendrer encore bien d'autres. Par exemple dans $\texttt{INT}_3$, la fonction :

$$\texttt{ajout} \circ (\texttt{proj}_1, \texttt{succ} \circ \texttt{proj}_2) : \texttt{Int} \times \texttt{Int} \to \texttt{Int}$$

dont l'interprétation dans M_Z est l'application :

$$M(\texttt{ajout} \circ (\texttt{proj}_1, \texttt{succ} \circ \texttt{proj}_2)) = + \circ (p_1, s \circ p_2) : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} : (x, y) \mapsto +(x, s(y)) = x + (y+1) \ ,$$

et la fonction :

$$\texttt{succ} \circ \texttt{ajout} \circ (\texttt{proj}_1, \texttt{proj}_2)$$

dont l'interprétation dans M_Z est l'application :

$$M(\texttt{succ} \circ \texttt{ajout} \circ (\texttt{proj}_1, \texttt{proj}_2)) = s \circ +(p_1, p_2) : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} : (x, y) \mapsto s(+(x, y)) = (x + y) + 1 \ .$$

Dans M_Z on a :

$$\text{pour tout } (x, y) \in \mathbb{Z} \times \mathbb{Z} \quad + (x, s(y)) = s(+(x, y)) \ .$$

Si on ne s'intéresse qu'aux modèles de $\texttt{INT}_3$ qui interprètent les deux fonctions $\texttt{ajout} \circ (\texttt{proj}_1, \texttt{succ} \circ \texttt{proj}_2)$ et $\texttt{succ} \circ \texttt{ajout} \circ (\texttt{proj}_1, \texttt{proj}_2)$ par deux applications égales, on considère la spécification $\texttt{INT}_4$:

Spécification $\texttt{INT}_4$:
Enrichit : $\texttt{INT}_3$;
Equation : $\texttt{ajout} \circ (\texttt{proj}_1, \texttt{succ} \circ \texttt{proj}_2) \equiv \texttt{succ} \circ \texttt{ajout} \circ (\texttt{proj}_1, \texttt{proj}_2)$.

SYNTAXE	produit potentiel	paire de fonctions
SEMANTIQUE	produit cartésien	paire d'applications

Remarque. Si on s'intéresse à des fonctions à n arguments, avec $n \geq 3$, on procède de façon analogue, mais à la place de produits potentiels binaires on utilise des produits potentiels *n-aires*, et à la place de la règle de construction de paires on a une règle de construction de *n-uplets*.

1.4 Constantes

On aimerait bien aussi ajouter dans la spécification un symbole, disons `zero`, pour représenter l'entier 0 dans le modèle M_Z. Un tel symbole est une *constante*, on va le voir comme une fonction particulière.

Sur les ensembles, on note $\{*\}$ un ensemble avec un unique élément, c'est-à-dire un *singleton* (le nom de son unique élément n'a aucune importance). Alors, on identifie l'élément 0 de l'ensemble $\mathbb{Z}$ avec l'application $* \mapsto 0$ de $\{*\}$ vers $\mathbb{Z}$:

$$0 : \{*\} \to \mathbb{Z} : * \mapsto 0 \ .$$

Dans la syntaxe, on procède de façon analogue. On construit une spécification $\texttt{INT}_5$ en enrichissant $\texttt{INT}_4$:

Spécification $\texttt{INT}_5$:
Enrichit : $\texttt{INT}_4$;
Type : `Void`;
Fonction : $\texttt{zero} : \texttt{Void} \to \texttt{Int}$.

On décide que lorsqu'un type s'appelle `Void`, son interprétation est forcément un singleton. Par conséquent, l'interprétation d'une fonction de source `Void`, comme `zero`, est une application constante, c'est-à-dire un élément.

Remarque : Les contraintes de singleton servent à **deux** choses. D'une part, comme on l'a vu, à imposer des contraintes aux modèles. D'autre part, à engendrer de nouvelles fonctions et de nouvelles équations, donc à *raisonner* et *calculer* à partir de la spécification. La règle fondamentale est l'*écrasement*, qu'on va voir maintenant.

Dans la spécification $\texttt{INT}_5$, la règle d'écrasement dit que :
SI on a un type X
ALORS on peut ajouter une fonction $!_X : X \to \texttt{Void}$.

Par conséquent, si M est un modèle de $\texttt{INT}_4$, il y a une seule façon d'interpréter $!_X$:

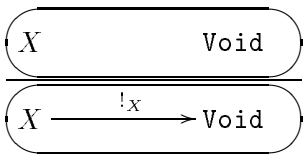
$$M(!_X) : M(X) \to M(\texttt{Void}) : x \mapsto * \ .$$

Par exemple, on peut ajouter une fonction $!_{\texttt{Int}} : \texttt{Int} \to \texttt{Void}$, dont l'unique interprétation dans M_Z est l'application $x \mapsto *$.

Sous forme de "fraction", cette règle s'écrit :

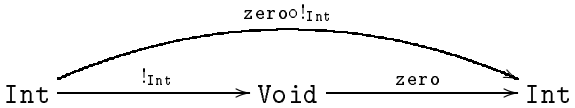
$$\frac{X}{!_X : X \to \texttt{Void}} \ .$$

On peut aussi la dessiner :

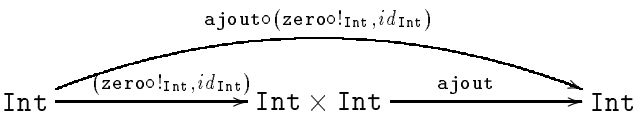


On peut utiliser $!_X$ pour engendrer de nouvelles fonctions.

Par exemple, la fonction :



est interprétée dans M_Z comme l'application $x \mapsto 0 : \mathbb{Z} \to \mathbb{Z}$. Cette fonction est utile pour former la fonction :



qui elle-même est interprétée dans M_Z comme l'application $x \mapsto 0 + x : \mathbb{Z} \to \mathbb{Z}$, et qui peut servir par exemple à traduire la propriété :

$$\text{pour tout } x \in \mathbb{Z} \quad 0 + x = x \ ,$$

en ajoutant l'équation :

$$\texttt{ajout} \circ (\texttt{zero} \circ !_{\texttt{Int}}, id_{\texttt{Int}}) \equiv id_{\texttt{Int}} : \texttt{Int} \to \texttt{Int} \ .$$

Parmi les fonctions engendrées par S, une *fonction constante* est une fonction de source `Void`. On l'appelle aussi un *terme clos* de S.

Comme on l'a déjà vu, les fonctions constantes jouent un rôle important. En effet, une fonction constante $t : \texttt{Void} \to Y$ est interprétée comme une application constante $M(f) : \{*\} \to M(X)$, c'est-à-dire comme un élément $M(f)(*)$ de $M(X)$.

SYNTAXE	type `Void`	fonction constante ou terme clos
SEMANTIQUE	singleton	élément

1.5 Programmes et calculs

On va définir des notions de programme et de calcul relatives à une spécification équationnelle S. Ces notions sont très limitées par rapport aux langages utilisés habituellement. Cependant elles forment un excellent point de départ pour étudier les programmes et les calculs en un sens plus général. Dans les chapitres qui suivent, on va voir quelques généralisations.

On définit un *programme* comme une fonction constante engendrée par S. Par exemple, voici quelques programmes de la spécification $\texttt{INT}_5$:

$$\texttt{zero},\ \texttt{succ} \circ \texttt{zero},\ \texttt{ajout} \circ (\texttt{succ}, \texttt{pred}) \circ \texttt{zero},\ \ldots$$

La relation de congruence définit une relation entre programmes. Par exemple dans $\mathtt{INT}_5$:

$$\mathtt{zero} \equiv \mathtt{ajout} \circ (\mathtt{succ}, \mathtt{pred}) \circ \mathtt{zero} \ .$$

On définit une *valeur* comme un programme particulièrement "simple" :
– une valeur est un programme particulier : il s'agit là d'une idée générale : résoudre un problème c'est le transformer un problème "équivalent" et tellement "simple" qu'on peut l'appeler "solution" ; de façon analogue, résoudre un système linéaire de Cramer c'est le transformer en un système linéaire équivalent et diagonal ;
– une valeur est un programme "simple" : à vrai dire, c'est seulement dans des situations particulières qu'on sait dire précisément ce que signifie "simple" ; de façon analogue, pour les systèmes linéaires de Cramer, un système simple est un système diagonal.

Un programme P *a pour* valeur V si P et V sont congrus : $P \equiv V$. Puisque la congruence respecte le type des arguments et du résultat, le programme et sa valeur sont parallèles : si $P : X \to Y$, alors $V : X \to Y$.

Par exemple, considérons encore une autre spécification $\mathtt{INT}_6$ "des entiers" :

Spécification $\mathtt{INT}_6$:
Types : `Int`, `Void`;
Fonctions : $\mathtt{succ} : \mathtt{Int} \to \mathtt{Int}$, $\mathtt{pred} : \mathtt{Int} \to \mathtt{Int}$, $\mathtt{zero} : \mathtt{Void} \to \mathtt{Int}$;
Equations : $\mathtt{pred} \circ \mathtt{succ} \equiv id_{\mathtt{Int}}$, $\mathtt{succ} \circ \mathtt{pred} \equiv id_{\mathtt{Int}}$.

Cette spécification a un modèle "des entiers" M_Z.

Alors on peut décider que les programmes suivants sont des valeurs :

$$\mathtt{zero},\ \mathtt{succ}^n \circ \mathtt{zero}\ (n \geq 1)\ ,\ \mathtt{pred}^n \circ \mathtt{zero}\ (n \geq 1)\ ,$$

où on note $f^n = f \circ f \circ \ldots \circ f$ (n fois). En effet ces programmes ne sont pas congrus entre eux, parce qu'ils ont des interprétations distinctes dans le modèle M_Z. De plus, sur cet exemple très simple, on peut montrer que tout programme est congru à une valeur.

Les équations de $\mathtt{INT}_6$ engendrent la congruence :

$$\mathtt{pred} \circ \mathtt{pred} \circ \mathtt{succ} \circ \mathtt{succ} \circ \mathtt{succ} \circ \mathtt{zero} \equiv \mathtt{succ} \circ \mathtt{zero} : \mathtt{Void} \to \mathtt{Int}\ ,$$

ce qui assure que dans le modèle M_Z :

$$p(p(s(s(s(0))))) = s(0)\ \ldots$$

Soit P un programme dans une spécification S.
On *prouve* que P a pour valeur V si on réussit à engendrer l'équation $P \equiv V$ à partir des équations de S. D'après Dowek ("La logique", coll. Dominos, Flammarion), chercher une preuve s'apparente parfois à chercher une aiguille dans une botte de foin.
On *calcule* la valeur V de P si on a une stratégie, c'est-à-dire un algorithme (une méthode systématique), en partant de P, pour engendrer l'équation $P \equiv V$, à partir des équations de S.
Ces questions de preuves et de calculs sont des questions fondamentales de l'informatique. Elles se posent souvent avec une définition plus générale des spécifications et des programmes, mais elles sont déjà complexes dans le cadre restreint de ce chapitre.

Dans ce cours, on va voir diverses façons de généraliser les notions de spécification et de programme, mais on ne va pas étudier les questions de preuve ni de calcul. Autrement dit, on va s'intéresser au cadre dans lequel les questions de preuve et de calcul peuvent avoir un sens, sans étudier directement ces questions.

1.6 Spécifications équationnelles

Les spécifications qu'on vient de décrire, formée de types, fonctions, produits potentiels et équations, s'appellent des *spécifications équationnelles*. Elles permettent de décrire de nombreuses structures. Par exemple, les monoïdes sont les modèles d'une spécification équationnelle assez facile à construire. De même pour les groupes ou les anneaux, mais pas pour les corps. Pour décrire les entiers, on a essayé de faire pour le mieux ci-dessus en construisant des spécifications de plus en plus précises, c'est-à-dire avec de moins en moins de modèles, qui ont toutes pour modèle "le modèle des entiers" M_Z. Cependant, un théorème difficile de logique prouve qu'on n'obtiendra jamais une spécification équationnelle dont l'unique modèle (à isomorphisme près) soit le modèle des entiers. Ce théorème dit que, en fait, bien que les entiers soient très simples, ils sont très difficiles à formaliser...

1.7 Exercices

Décrire une spécification des booléens, puis une spécification des "entiers machine", au sens des entiers `short` et `unsigned` de C, c'est-à-dire que ces nombres sont toujours positifs ou nuls et que les opérations suivent les lois de l'arithmétique modulo 2^{16}.

Décrire une spécification des naturels. Puis décrire une spécification des listes : listes de booléens, listes de naturels, listes sur un type X quelconque.

Dans chacun de ces exemples, écrire des programmes (au sens de ce chapitre) et déterminer leur valeur.

2 Si... alors... sinon... (29/09/03)

Dans ce chapitre on enrichit les notions de type et de fonction vues au chapitre 1, afin d'autoriser les "`si...alors...sinon...`", ou plus généralement les *branchements* ("`switch-case`" en C), dans les programmes.

Rappelons que les *produits potentiels* du chapitre 1 permettent de construire des paires (cas binaire) ou plus généralement des n-uplets. Ici on va introduire des *sommes potentielles* pour construire des "si... alors... sinon..." (cas binaire) ou plus généralement des branchements. De plus, on verra que ces sommes potentielles permettent d'ajouter des axiomes autres que des équations.

2.1 Branchements

On a vu qu'une *spécification équationnelle* est formée de types, fonctions, équations et produits potentiels, qui repésentent respectivement :

SYNTAXE	type	fonction	équation	produit potentiel
SEMANTIQUE	ensemble	application	égalité	produit cartésien

Considérons, sur l'ensemble $\mathbb{Z}$ des naturels, l'application "valeur absolue" :

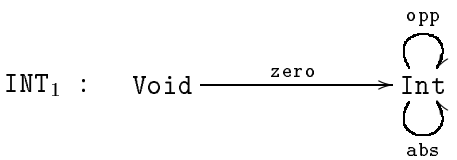
$$|\ldots| : \mathbb{Z} \to \mathbb{Z} : x \mapsto |x| \ ,$$

qui est définie par la propriété suivante :

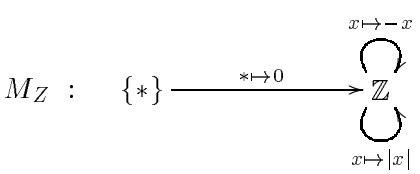
$$\text{pour tout } x \in \mathbb{N} \quad |x| = (\ \texttt{si}\ x \geq 0\ \texttt{alors}\ x\ \texttt{sinon}\ -x)\ .$$

Considérons une spécification équationnelle "des entiers" $\texttt{INT}_1$, qui comporte une fonction *opposé* $\texttt{opp} : \texttt{Int} \to \texttt{Int}$ et une fonction *valeur absolue* $\texttt{abs} : \texttt{Int} \to \texttt{Int}$:

Spécification $\texttt{INT}_1$:
Types : `Int`, `Void`;
Fonctions : `opp`, `abs` : `Int` → `Int`, `zero` : `Void` → `Int`.



Elle a encore un modèle "des entiers", qu'on note toujours M_Z :



On cherche maintenant à enrichir $\texttt{INT}_1$ pour spécifier la propriété de la valeur absolue, qui n'est pas une équation à cause du branchement " $\texttt{si}\ x \geq 0 \ldots$". On va faire cet enrichissement en deux temps.

Tout d'abord, il faut traduire le test "$x \geq 0$?", et pour cela on introduit la notion de somme de deux ensembles.

Pour tous ensembles *disjoints* X et Y, l'ensemble des x tels que x est dans X ou dans Y (puisque X et Y sont disjoints, x ne peut pas être à la fois dans X et dans Y) se note $X + Y$ et s'appelle la *réunion disjointe*, ou la *somme*, de X et Y.

Attention ! Ici on utilise, pour une opération sur les ensembles, le même mot "somme" et le même symbole "+", que pour une opération sur les nombres. Il ne faut pas les confondre, mais quand même il y a un rapport entre les deux : si les ensembles X et Y sont finis, alors l'ensemble $X + Y$ aussi, et $Card(X + Y) = Card(X) + Card(Y)$.

La somme $X + Y$ contient tous les éléments de X, tous ceux de Y, et rien d'autre. Elle est associée à deux inclusions :

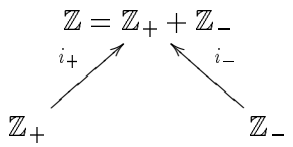
$$\texttt{incl}_1 : X \to X + Y \ , \ \texttt{incl}_2 : Y \to X + Y \ .$$

Notons $\mathbb{Z}_+$ l'ensemble des entiers ≥ 0, et $\mathbb{Z}_-$ l'ensemble des entiers < 0. Alors $\mathbb{Z} = \mathbb{Z}_+ + \mathbb{Z}_-$, et la propriété de l'application $|\ldots|$ peut s'exprimer ainsi :

$$\text{pour tout } x \in \mathbb{Z}_+ \quad |x| = x \ ,$$

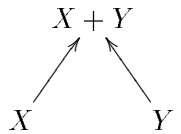
$$\text{et pour tout } x \in \mathbb{Z}_- \quad |x| = -x \ .$$

Les deux inclusions correspondant à la somme $\mathbb{Z} = \mathbb{Z}_+ + \mathbb{Z}_-$ sont notées i_+ et i_- :



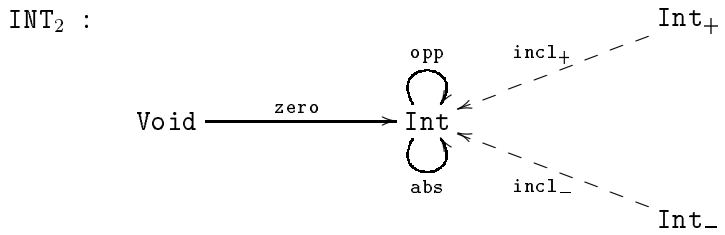
Au chapitre 1, on a défini les produits potentiels d'une spécification comme des cônes projectifs particuliers, qui sont dessinés d'une façon particulière (en pointillé). Un produit potentiel impose une contrainte sur les modèles (produit cartésien). Pour traduire la propriété de $|\ldots|$, on va ajouter à $\texttt{INT}_1$ une *somme potentielle.*

On appelle *cône inductif (binaire)* un couple de fonctions de même but mais pas forcément de même source. Et on appelle *sommet* du cône le but commun à ses deux fonctions, et *base* du cône la liste de leurs sources. Pour dire qu'un cône inductif doit devenir une somme, on l'appelle une *somme potentielle*. On note souvent $X + Y$ le sommet de la somme potentielle de base (X, Y) :



Specification $\texttt{INT}_2$:
Enrichit $\texttt{INT}_1$;
Types : $\texttt{Int}_+$, $\texttt{Int}_-$;
Fonctions : $\texttt{incl}_+ : \texttt{Int}_+ \to \texttt{Int}$, $\texttt{incl}_- : \texttt{Int}_- \to \texttt{Int}$;
Somme potentielle : (sommet $\texttt{Int} = \texttt{Int}_+ + \texttt{Int}_-$, inclusions $\texttt{incl}_+$, $\texttt{incl}_-$).

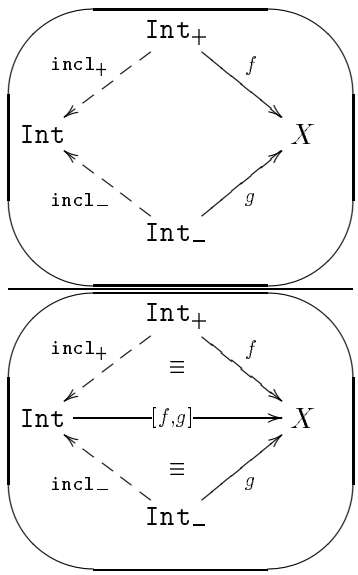
Pour préciser que, parmi tous les cônes inductifs, certains sont des sommes potentielles, on décide de représenter les inclusions des sommes potentielles par des flèches tiretées :



Remarque. Les sommes potentielles servent à **deux** choses. D'une part, comme on l'a vu, les sommes potentielles servent à imposer des contraintes aux modèles. D'autre part, les sommes potentielles servent aussi à former de nouvelles fonctions, et à *raisonner* et *calculer* à partir de la spécification. La règle fondamentale est la *construction de branchements*, qu'on va voir maintenant.

Dans la spécification $\texttt{INT}_2$, la règle de construction de branchements dit que :
SI on a un type X et deux fonctions $f : \texttt{Int}_+ \to X$ et $g : \texttt{Int}_- \to X$
ALORS on peut ajouter une fonction $[f, g] : \texttt{Int} \to X$
avec les équations $[f, g] \circ \texttt{incl}_+ \equiv f$ et $[f, g] \circ \texttt{incl}_- \equiv g$.
La notation $[\ldots, \ldots]$ n'est pas standard, contrairement à $(\ldots, \ldots)$.

On peut dessiner cette règle :



Comme d'habitude, cette règle $\frac{H}{C}$ signifie :

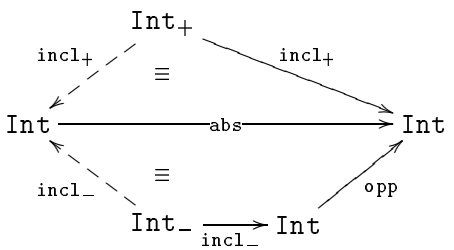
Toute occurrence de H dans une spécification peut être complétée par une occurrence de C, sans modifier les modèles.

Maintenant, dans $\texttt{INT}_2$, pour traduire la définition :

$$|x| =_{def} (\ \texttt{si}\ x \geq 0\ \texttt{alors}\ x\ \texttt{sinon}\ -x)\ ,$$

il suffit de poser :

$$\mathtt{abs} = [\mathtt{incl}_+, \mathtt{opp} \circ \mathtt{incl}_-] \, .$$



Les nouvelles fonctions $[f, g]$ permettent de construire beaucoup de termes comportant des "si...alors...sinon...".

SYNTAXE	somme potentielle	branchement de fonctions
SEMANTIQUE	somme (réunion disjointe)	branchement d'applications

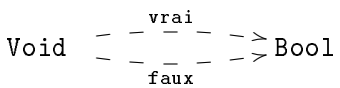
De plus, l'exemple de la valeur absolue montre qu'avec des sommes potentielles et des équations, on peut traduire des axiomes qui ne sont pas des équations.

Remarque. Si on s'intéresse à des branchements à n cas, avec $n \geq 3$, on procède de façon analogue, mais à la place de sommes potentielles binaires on utilise des sommes potentielles *n-aires*, et on généralise la règle de construction de branchements.

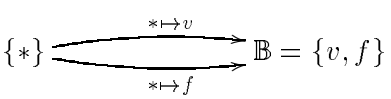
2.2 Les booléens

Il est maintenant possible de construire une spécification `BOOL` dont l'unique modèle détermine l'ensemble des booléens. Remarque : lorsqu'on parle de "unique modèle", c'est toujours à isomorphisme près ; en effet l'ensemble des booléens est toujours un ensemble à deux éléments, mais ces éléments peuvent peuvent être notés v et f, ou 1 et 0, ou encore *true* et *false*, etc. Ici on note $\mathbb{B} = \{v, f\}$. Une spécification très simple des booléens est:

Spécification $\mathtt{BOOL}_1$:
Types : `Void`, `Bool`;
Fonctions : $\mathtt{vrai} : \mathtt{Void} \to \mathtt{Bool}$, $\mathtt{faux} : \mathtt{Void} \to \mathtt{Bool}$;
Somme potentielle : (sommet $\mathtt{Bool} = \mathtt{Void} + \mathtt{Void}$, inclusions `vrai`, `faux`).



L'unique modèle M_B de $\mathtt{BOOL}_1$ est bien :



On peut ensuite obtenir les fonctions `non`, `et`, `ou`, etc, par une simple construction de branchements. Par exemple pour la négation, on pose : :

$$\mathtt{non} = [\mathtt{faux}, \mathtt{vrai}] \ : \ \mathtt{Bool} \to \mathtt{Bool} \ ,$$

donc l'interprétation par M_B est l'application :

$$\neg \ : \ \mathbb{B} \to \mathbb{B} =_{def} \ (\ x \ \mapsto \ \ \mathtt{si}\ x = v\ \mathtt{alors}\ f\ \mathtt{sinon}\ v \ ) \ .$$

On peut aussi, en réunissant les deux spécifications $\mathtt{INT}_2$ et $\mathtt{BOOL}_1$, obtenir la fonction :

$$\mathtt{positif} = [\mathtt{vrai} \circ !_{\mathtt{Int}_+}, \mathtt{faux} \circ !_{\mathtt{Int}_-}] \ : \ \mathtt{Int} \to \mathtt{Bool} \ .$$

Dans le modèle formé à partir de M_Z et M_B, cette fonction s'interprète comme le *prédicat* (c'est-à-dire l'application à valeurs booléennes) :

$$\geq 0? \ : \ \mathbb{Z} \to \mathbb{B} =_{def} \ (\ x \ \mapsto \ \ \mathtt{si}\ x \geq 0\ \mathtt{alors}\ v\ \mathtt{sinon}\ f \ ) \ .$$

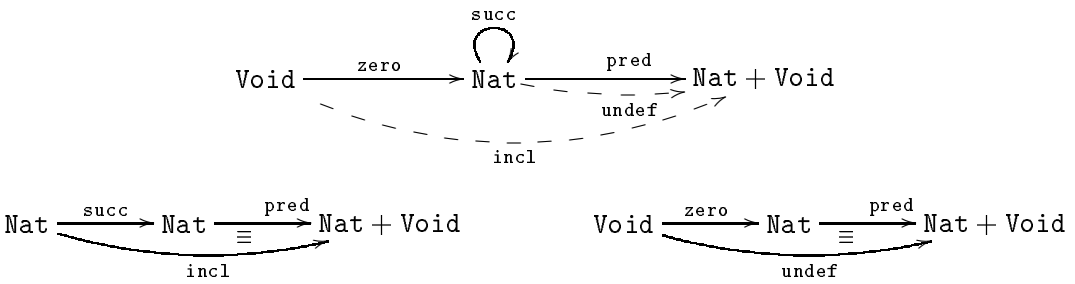
2.3 Fonctions partielles

Une façon de traiter d'une application partielle est d'ajouter un nouvel élément, disons ε, à son ensemble d'arrivée, et de prolonger l'application de façon à ce qu'elle retourne ε aux valeurs où elle n'était pas définie. Par exemple, sur les naturels, on peut définir une fonction prédécesseur, non définie en 0, en posant :

$$\mathtt{pred} \ : \ \mathbb{N} \to \mathbb{N} + \{\varepsilon\} =_{def} \ (\ x \mapsto \ \ \mathtt{si}\ x \neq 0\ \mathtt{alors}\ x - 1\ \mathtt{sinon}\ \varepsilon \ ) \ .$$

Cette approche est facile à formaliser dès qu'on dispose de sommes potentielles. Par exemple :

Spécification $\mathtt{NAT}_1$:
Types : Nat, Void, Nat + Void;
Fonctions : succ : Nat → Nat, zero : Void → Nat, pred : Nat → Nat + Void, ;
Somme potentielle : (sommet Nat + Void, inclusions incl, undef),
Equations : pred ∘ succ ≡ incl, pred ∘ zero ≡ undef.



2.4 Programmes et calculs

D'un point de vue théorique, on généralise facilement les notions de programme et de calcul du chapitre 1, de façon à autoriser les branchements dans les programmes. Il suffit de partir d'une spécification S qui peut comporter des sommes potentielles.

Comme au chapitre 1, un *programme* sur S est un terme clos de S (c'est-à-dire une fonction constante engendrée par S), la relation de congruence définit une relation d'équivalence entre programmes, une *valeur* est un programme particulièrement "simple", et un programme P *a pour* valeur V si $P \equiv V$, ce qui implique que P et V sont parallèles. Les notions de *preuve* et de *calcul* se généralisent aussi, en théorie, sans problème.

Mais de plus ici, on impose qu'une valeur ne contienne aucun branchement. Cela signifie que la spécification doit contenir suffisamment d'information pour qu'on puisse faire disparaître tous les branchements en utilisant les équations.

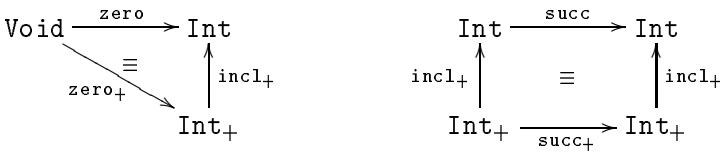
Par exemple, dans la spécification $\texttt{INT}_2$, avec un choix "raisonnable" des valeurs, la valeur du programme $\texttt{abs} \circ \texttt{succ} \circ \texttt{zero}$ devrait être $\texttt{succ} \circ \texttt{zero}$. Mais $\texttt{abs} \circ \texttt{succ} \circ \texttt{zero}$ n'est pas congru à $\texttt{succ} \circ \texttt{zero}$ dans $\texttt{INT}_2$, car cette spécification "des entiers" est trop incomplète.

Pour comprendre comment enrichir $\texttt{INT}_2$, on raisonne d'abord sur les entiers : on sait que $0 \in \mathbb{Z}_+$ et que $s : \mathbb{Z}_+ \to \mathbb{Z}_+$, d'où $s(0) \in \mathbb{Z}_+$, et donc $|s(0)| = s(0)$.

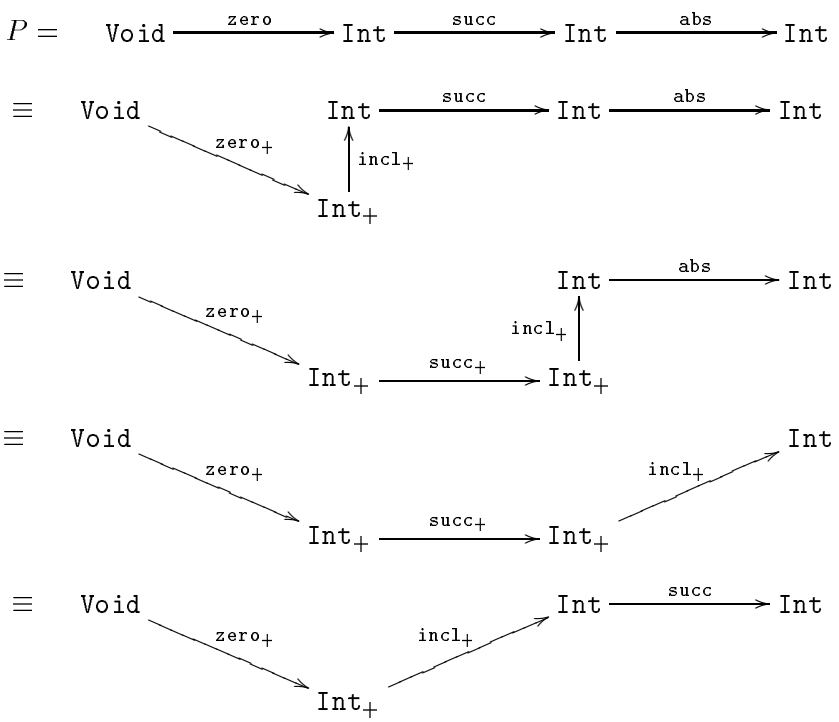
Par conséquent, on enrichit la spécification $\texttt{INT}_2$ de la façon suivante, qui impose de nouvelles contraintes aux modèles.

Spécification $\texttt{INT}_3$:
Enrichit : $\texttt{INT}_2$;
Fonctions : $\texttt{zero}_+ : \texttt{Void} \to \texttt{Int}_+$, $\texttt{succ}_+ : \texttt{Int}_+ \to \texttt{Int}_+$;
Equations : $\texttt{incl}_+ \circ \texttt{zero}_+ \equiv \texttt{zero}$, $\texttt{incl}_+ \circ \texttt{succ}_+ \equiv \texttt{succ} \circ \texttt{incl}_+$.

Ces équations se dessinent ainsi :



Alors la suite de congruences qui mène à l'évaluation du programme $P = \texttt{abs} \circ \texttt{succ} \circ \texttt{zero}$ en $V = \texttt{succ} \circ \texttt{zero}$ se dessine facilement :



$$\equiv \quad \texttt{Void} \xrightarrow{\texttt{zero}} \texttt{Int} \xrightarrow{\texttt{succ}} \texttt{Int} = V$$

On peut remarquer que les deux équations de $\texttt{INT}_3$ ont été utilisées deux fois, une fois dans chaque sens. On peut aussi remarquer que pour aller d'un chemin de longueur 3 à un chemin de longueur 2, on est passé par des chemins de longueur 4. Ce tout petit exemple fournit donc des indices sur la difficulté qu'il peut y avoir à automatiser le processus d'évaluation...

2.5 Exercices

Lorsque des prédicats sont partiellement définis, les connecteurs "et" et "et puis" peuvent retourner des résultats différents. Exprimer cela dans une spécification.

Choisir une spécification équationnelle "des naturels", et y ajouter une somme potentielle pour préciser que tout naturel est soit nul, soit un successeur, et pas les deux. Est-ce cette spécification a des modèles autres que le modèle "des naturels" ?

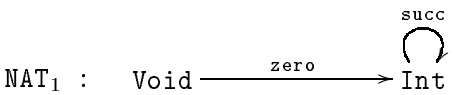
3 Récurrence (14/10/03)

3.1 Comment spécifier les naturels ?

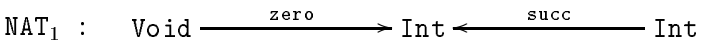
Considérons la spécification équationnelle "des naturels" suivante :

Spécification $\texttt{NAT}_1$:
Types : `Nat`, `Void`;
Fonctions : `zero` : `Void` $\to$ `Nat`, `succ` : `Nat` $\to$ `Nat`.

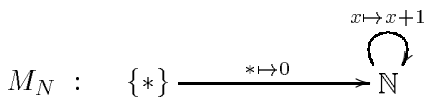
ou bien :



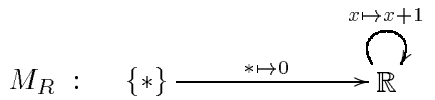
ou bien encore :

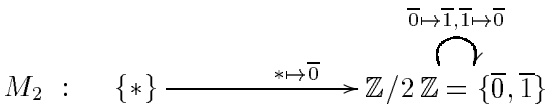


Cette spécification a pour modèle "des naturels" :



Mais elle a aussi de nombreux autres modèles : tout ensemble muni d'un élément et d'une opération unaire fournit un modèle de $\texttt{NAT}_1$. Parmi les modèles qui interprètent `zero` comme un 0 et `succ` comme $x \mapsto x+1$, figurent le modèle M_R "des réels" et le modèle M_2 "des entiers modulo 2" :





On cherche maintenant à enrichir $\texttt{NAT}_1$ de façon à ce que M_N soit son unique modèle (à isomorphisme près). Il faut pour cela traduire, au niveau de la syntaxe, une propriété qui caractérise l'ensemble $\mathbb{N}$. On va considérer la propriété de récurrence, sous la forme suivante :

> Pour tout ensemble A, tout élément $a \in A$ et toute application $\alpha : A \to A$, il y a exactement une application $\varphi : \mathbb{N} \to A$ telle que $\varphi(0) = a$ et $\varphi(x+1) = \alpha(\varphi(x))$.

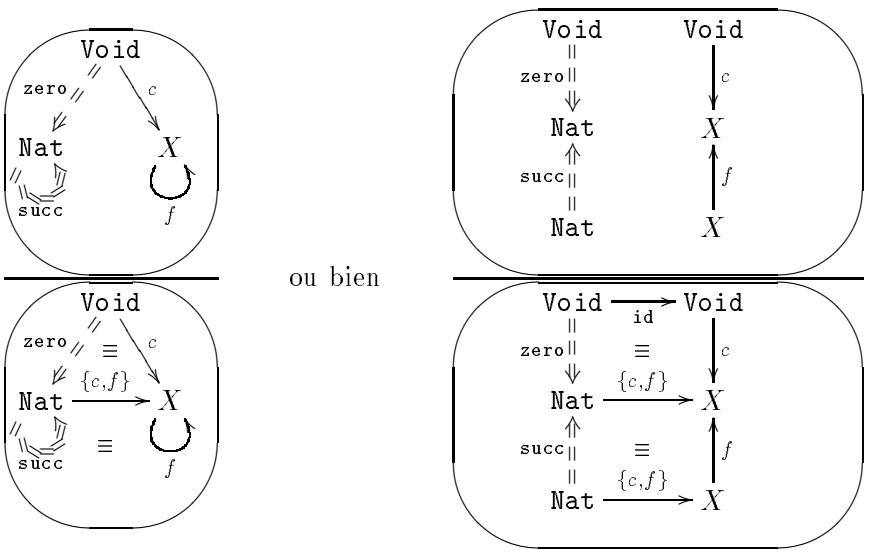
Par exemple, lorsque $A = \mathbb{N}$, $a = 1$ et $\alpha(y) = 2\,y$, il y a exactement une application $\varphi : \mathbb{N} \to \mathbb{N}$ telle que $\varphi(0) = 1$ et $\varphi(x+1) = \varphi(x) + 2$: on montre facilement, par récurrence sur n, que c'est l'application $\varphi(n) = 2\,n$.

Au chapitre 1 on a introduit des contraintes de produit, et au chapitre 2 des contraintes de somme. De façon analogue, on va introduire ici une nouvelle forme de contrainte pour traduire cette propriété de récurrence. Cette contrainte porte sur l'ensemble de la spécification $\texttt{NAT}_1$, elle

s'exprime par la règle suivante :

SI on a un type X et deux fonctions $c : \mathtt{Void} \to X$ et $f : X \to X$
ALORS on peut ajouter une fonction $\{c, f\} : \mathtt{Nat} \to X$
avec les équations $\{c, f\} \circ \mathtt{zero} \equiv c$ et $\{c, f\} \circ \mathtt{succ} \equiv f \circ \{c, f\}$.

On peut dessiner cette règle, sous deux formes équivalentes :



On peut maintenant enrichir la spécification $\mathtt{NAT}_1$:

Spécification $\mathtt{NAT}_2$:
Enrichit : $\mathtt{NAT}_1$;
Contrainte d'initialité : $\mathtt{NAT}_1$ est initial.

ou bien :

$$\mathtt{NAT}_2 \ : \quad \mathtt{Void} \overset{\mathtt{zero}}{\Longrightarrow} \mathtt{Int} \circlearrowleft \mathtt{succ}$$

ou bien encore :

$$\mathtt{NAT}_2 \ : \quad \mathtt{Void} \overset{\mathtt{zero}}{\Longrightarrow} \mathtt{Int} \overset{\mathtt{succ}}{\Longleftarrow} \mathtt{Int}$$

En fait, la spécification $\mathtt{NAT}_2$ a un seul modèle (à isomorphisme près), à savoir M_N.

Remarque. Les contraintes d'initialité servent à **deux** choses. D'une part, à imposer des contraintes aux modèles : ici, $\mathtt{NAT}_2$ a pour seul modèle M_N. D'autre part, à former de nouvelles fonctions (les fonctions $\{c, f\}$), et à *raisonner* et *calculer* à partir de la spécification : l'existence de $\{c, f\}$ permet de faire des définitions par récurrence, et son unicité permet de faire des preuves par récurrence.

Par exemple, posons $X = \mathtt{Nat}$, $c = \mathtt{zero} : \mathtt{Void} \to \mathtt{Nat}$ et $f = \mathtt{succ} \circ \mathtt{succ} : \mathtt{Nat} \to \mathtt{Nat}$. La règle fournit une nouvelle opération $\{c, f\} : \mathtt{Nat} \to \mathtt{Nat}$ telle que $\{c, f\} \circ \mathtt{zero} \equiv \mathtt{zero}$ et $\{c, f\} \circ \mathtt{succ} \equiv \mathtt{succ} \circ \mathtt{succ} \circ \{c, f\}$. Dans le modèle M_N, la fonction $\{c, f\}$ est interprétée par l'application $M(\{c, f\}) : \mathbb{N} \to \mathbb{N}$ telle que $M(\{c, f\})(0) = 0$ et $M(\{c, f\})(x+1) = M(\{c, f\})(x)+2$. Autrement dit, $M(\{c, f\})$ est l'application φ déjà rencontrée, telle que $\varphi(n) = 2\,n$.

3.2 Termes clos : l'exemple des naturels

On peut reprendre l'exemple précédent sous un autre angle, en oubliant $\mathtt{NAT}_2$. La spécification équationnelle $\mathtt{NAT}_1$ a de nombreux modèles, parmi eux on a déjà défini le modèle M_N "des naturels", et maintenant on va définir le modèle T "des termes clos".

Le modèle T

On note $T(\mathtt{Nat})$ l'ensemble des termes clos de $\mathtt{NAT}_1$ de but $\mathtt{Nat}$:

$$T(\mathtt{Nat}) = \{\mathtt{zero}, \mathtt{succ} \circ \mathtt{zero}, \mathtt{succ} \circ \mathtt{succ} \circ \mathtt{zero}, \ldots\} = \{\mathtt{succ}^n \circ \mathtt{zero} \mid n \in \mathbb{N}\} .$$

On note $T(\mathtt{succ})$ l'application de $T(\mathtt{Nat})$ dans lui-même, définie par $T(\mathtt{succ})(\mathtt{succ}^n \circ \mathtt{zero}) = \mathtt{succ} \circ \mathtt{succ}^n \circ \mathtt{zero}$:

$$T(\mathtt{succ}) : T(\mathtt{Nat}) \to T(\mathtt{Nat}) : \mathtt{succ}^n \circ \mathtt{zero} \mapsto \mathtt{succ}^{n+1} \circ \mathtt{zero} .$$

On note $T(\mathtt{Void})$ l'ensemble des termes clos de $\mathtt{NAT}_1$ de but $\mathtt{Void}$:

$$T(\mathtt{Void}) = \{\mathtt{id}_{\mathtt{Void}}\} .$$

On note $T(\mathtt{zero})$ l'application de $T(\mathtt{Void})$ vers $T(\mathtt{Nat})$, définie par $T(\mathtt{zero})(\mathtt{id}_{\mathtt{Void}}) = \mathtt{zero} \circ \mathtt{id}_{\mathtt{Void}}$:

$$T(\mathtt{zero}) : T(\mathtt{Void}) \to T(\mathtt{Nat}) : \mathtt{id}_{\mathtt{Void}} \mapsto \mathtt{zero} .$$

Ainsi, on vient de définir un modèle T de $\mathtt{NAT}_1$.

Les modèles T et M_N

En associant à tout $n \in \mathbb{N}$ le terme $\mathtt{succ}^n \circ \mathtt{zero}$, on définit une bijection entre l'ensemble $\mathbb{N} = M_N(\mathtt{Nat})$ et l'ensemble $T(\mathtt{Nat})$:

$$\begin{array}{ccc} M_N(\mathtt{Nat}) = \mathbb{N} & & n \\ \cong \updownarrow & & \updownarrow \\ T(\mathtt{Nat}) & & \mathtt{succ}^n \circ \mathtt{zero} \end{array}$$

L'application $M_N(\mathtt{succ})$, telle que $n \mapsto n+1$, correspond par cette bijection à l'application $T(\mathtt{succ})$:

$$\begin{array}{ccc} M_N(\mathtt{Nat}) = \mathbb{N} & \xrightarrow{M_N(\mathtt{succ})} & M_N(\mathtt{Nat}) = \mathbb{N} \\ \cong \updownarrow & \equiv & \updownarrow \cong \\ T(\mathtt{Nat}) & \xrightarrow{T(\mathtt{succ})} & T(\mathtt{Nat}) \end{array}$$

En associant à $*$ le terme $\mathtt{id}_{\mathtt{Void}}$, on définit une bijection entre l'ensemble $\{*\} = M_N(\mathtt{Void})$ et l'ensemble $T(\mathtt{Void})$:

$$\begin{array}{ccc} M_N(\mathtt{Void}) = \{*\} & & * \\ \cong \updownarrow & & \updownarrow \\ T(\mathtt{Void}) & & \mathtt{id}_{\mathtt{Void}} \end{array}$$

L'application $M_N(\texttt{zero})$, telle que $* \mapsto 0$, correspond par cette bijection à l'application $T(\texttt{zero})$:

$$\begin{array}{ccc} M_N(\texttt{Void}) = \{*\} & \xrightarrow{M_N(\texttt{zero})} & M_N(\texttt{Nat}) = \mathbb{N} \\ \cong \uparrow\downarrow & \equiv & \uparrow\downarrow \cong \\ T(\texttt{Void}) & \xrightarrow{T(\texttt{zero})} & T(\texttt{Nat}) \end{array}$$

Tout cela signifie que les deux modèles M_N et T de $\texttt{NAT}_1$ sont *isomorphes.*

Or M_N est aussi l'unique modèle de $\texttt{NAT}_2$. Par conséquent, on vient de montrer que l'unique modèle de $\texttt{NAT}_2$ est le modèle des termes clos de $\texttt{NAT}_1$.

3.3 Termes clos

Définition 3.1 Une *signature* est une spécification équationnelle sans équation.

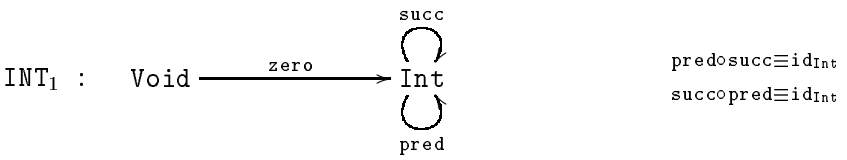
Les résultats qu'on vient de voir sur les naturels se généralisent à toutes les signatures :

Théorème 3.1 *Soit S_1 une signature, alors les termes clos de S_1 forment un modèle T de S_1. Soit S_2 la spécification obtenue en ajoutant à S_1 la contrainte "S_1 est initial", alors T est l'unique modèle de S_2, à isomorphisme près.*

Lorsque S_1 a des équations, il faut en tenir compte avant de généraliser ce théorème.

Par exemple, considérons la spécification "des entiers" suivante :

Spécification $\texttt{INT}_1$:
Types : Int, Void;
Fonctions : $\texttt{zero} : \texttt{Void} \to \texttt{Int}$, $\texttt{succ}, \texttt{pred} : \texttt{Int} \to \texttt{Int}$.
Equations : $\texttt{pred} \circ \texttt{succ} \equiv \texttt{id}_{\texttt{Int}}$, $\texttt{succ} \circ \texttt{pred} \equiv \texttt{id}_{\texttt{Int}}$.

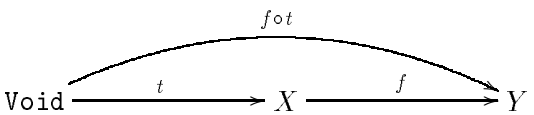


Pour chaque type X de S_1, on définit $T(X)$ comme l'ensemble de tous les termes clos de S_1 de but X, c'est-à-dire l'ensemble de toutes les fonctions de source Void et de but X qui peuvent être engendrées à partir de S_1 :

$$T(X) = \{t : \texttt{Void} \to X \mid t \text{ terme de } S_1\} .$$

Pour chaque fonction $f : X \to Y$ de S_1, on définit $T(f) : T(X) \to T(Y)$ comme l'application :

$$T(f) : T(X) \to T(Y) : t \mapsto f \circ t .$$



Si $f \equiv g : X \to Y$ est une équation de S_1 et si $t : \texttt{Void} \to X$ est un terme clos, alors $f \circ t \equiv g \circ t : \texttt{Void} \to Y$. Mais (dès que $f \neq g$) les deux termes $f \circ t$ et $g \circ t$ sont distincts, par conséquent les applications $T(f)$ et $T(g)$ sont différentes.

Donc T ne vérifie **pas** les équations de S_1, par conséquent T forme **pas** un modèle de S_1, dès que S_1 a des équations.

Par exemple $\texttt{pred} \circ \texttt{succ}$ et $\texttt{id}_{\texttt{Int}}$ sont deux termes congrus de S_1, mais leurs interprétations par T sont deux applications distinctes : en effet, en considérant par exemple la valeur de ces deux applications en $\texttt{zero}$, on obtient $T(\texttt{pred} \circ \texttt{succ})(\texttt{zero}) = \texttt{pred} \circ \texttt{succ} \circ \texttt{zero}$ et $T(\texttt{id}_{\texttt{Int}})(\texttt{zero}) = \texttt{zero}$, or les termes clos $\texttt{pred} \circ \texttt{succ} \circ \texttt{zero}$ et $\texttt{zero}$ sont différents (ils sont seulement congrus).

3.4 Termes clos modulo les congruences

Sur l'ensemble $T(X)$ des termes clos de S_1, les congruences de S_1, c'est-à-dire les équations engendrées par S_1, forment une *relation d'équivalence*. L'ensemble des classes d'équivalence est noté :

$$I(X) = T(X)/\equiv \ .$$

Cela signifie qu'un élément x de $I(X)$ est une *classe d'équivalence* de $T(X)$ pour la relation $\equiv$. Autrement dit, x est un sous-ensemble de $T(X)$ qui vérifie deux propriétés :
– si deux termes clos t et u sont dans x alors $t \equiv u$,
– si t est dans x et u n'y est pas, alors $t \not\equiv u$.

Attention ! On a $x \in I(X)$ mais $x \subseteq T(X)$, autrement dit x est un *élément* de $I(X)$ mais x est un *sous-ensemble* de $T(X)$.

On dit que x est *la classe de* t dès que $t \in x$, et on note $x = C(t)$; à un terme clos t correspond une seule classe $x = C(t)$, mais à une classe x il peut correspondre de nombreux termes clos $t, u, \ldots$ tels que $x = C(t) = C(u) = \ldots$ On a :

$$C(t) = C(u) \iff t \equiv u \ .$$

Par exemple, pour la spécification $\texttt{INT}_1$, on peut poser $x = C(\texttt{zero})$, et alors $x = C(\texttt{pred} \circ \texttt{succ} \circ \texttt{zero}) = C(\texttt{succ} \circ \texttt{pred} \circ \texttt{zero}) = \ldots$ En fait :

$$I(\texttt{Int}) = \{C(\texttt{zero})\} \cup \{C(\texttt{succ}^n \circ \texttt{zero}) \mid n > 0\} \cup \{C(\texttt{pred}^n \circ \texttt{zero}) \mid n > 0\} \cong \mathbb{Z}$$

$$\text{et } I(\texttt{Void}) = \{C(\texttt{id}_{\texttt{Void}})\} \cong \{*\} \ .$$

Soit $f : X \to Y$ un terme de S_1. On définit une application :

$$I(f) : I(X) \to I(Y)$$

en posant : soit x un élément de $I(X)$, et soit t tel que $x = C(t)$, alors $I(f)(x) = C(f \circ t)$; autrement dit :

$$I(f)(C(t)) = C(f \circ t) \ .$$

Il faut vérifier que cette définition est valide, c'est-à-dire que la valeur de $I(f)(x)$ ne dépend vraiment que de x, pas du choix de t dans la classe x. Il s'agit donc de montrer que :

si $C(t) = C(u)$ alors $C(f \circ t) = C(f \circ u)$,

ce qui revient à dire que :

si $t \equiv u$ alors $f \circ t \equiv f \circ u$,
ce qui est vrai par l'une des règles de congruence.

Par exemple, pour la spécification $\mathtt{INT}_1$, on a déjà vu que $I(\mathtt{Int}) \cong \mathbb{Z}$ et $I(\mathtt{Void}) \cong \{*\}$. Avec ces bijections, l'application $I(\mathtt{succ})$ correspond à $x \mapsto x+1 : \mathbb{Z} \to \mathbb{Z}$, l'application $I(\mathtt{pred})$ correspond à $x \mapsto x-1 : \mathbb{Z} \to \mathbb{Z}$, et l'application $I(\mathtt{zero})$ correspond à $* \mapsto 0 : \{*\} \to \mathbb{Z}$.

Il reste à montrer que les équations de S_1 deviennent, par I, des égalités. Soit $f \equiv g : X \to Y$ une équation de S_1, et soit $x = C(t) \in I(X)$. Alors $f \circ t \equiv g \circ t$, par une autre règle de congruence, donc $I(f)(x) = I(g)(x)$. Puisque c'est vrai pour tout élément x de $I(X)$, cela prouve l'égalité des deux applications : $I(f) = I(g)$. Par conséquent, I forme un modèle de S_1.

Par exemple, pour la spécification $\mathtt{INT}_1$, on a bien $I(\mathtt{pred} \circ \mathtt{succ}) = id_{\mathbb{Z}}$ et $I(\mathtt{succ} \circ \mathtt{pred}) = id_{\mathbb{Z}}$.

On vient de montrer la première partie du résultat suivant (on admet la seconde partie) :

Théorème 3.2 *Soit S_1 une spécification équationnelle, alors les termes clos de S_1 modulo les congruences de S_1 forment un modèle I de S_1.*
Soit S_2 la spécification obtenue en ajoutant à S_1 la contrainte "S_1 est initial", alors I est l'unique modèle de S_2, à isomorphisme près.

Définition 3.2 Avec ces notations, les modèles de S_2 sont appelés les *modèles initiaux* de S_1.

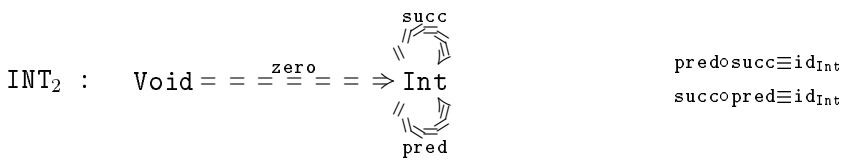
Donc le théorème3.2 dit que :

Théorème 3.3 *Soit S_1 une spécification équationnelle, alors les termes clos de S_1 modulo les congruences de S_1 forment un modèle initial de S_1.*

Ce résultat relie la *syntaxe* (les fonctions et équations) avec la *sémantique* (les modèles), en passant par les termes et les congruences engendrés.

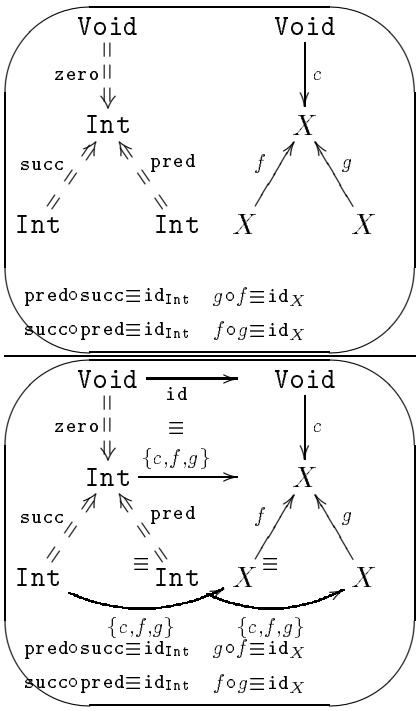
La seconde partie de ce théorème permet de généraliser la récurrence. Par exemple, dans le cas des entiers, la spécification $\mathtt{INT}_2$ est la suivante :

Spécification $\mathtt{INT}_2$:
Enrichit : $\mathtt{INT}_1$;
Contrainte d'initialité : $\mathtt{INT}_1$ est initial.



La contrainte d'initialité impose la règle suivante :
SI on a un type X et trois fonctions $c : \mathtt{Void} \to X$, $f : X \to X$ et $g : X \to X$
avec les équations $g \circ f \equiv \mathtt{id}_X$ et $f \circ g \equiv \mathtt{id}_X$
ALORS on peut ajouter une fonction $\{c, f, g\} : \mathtt{Int} \to X$

avec les équations $\{c,f\}\circ\texttt{zero} \equiv c$, $\{c,f\}\circ\texttt{succ} \equiv f\circ\{c,f\}$ et $\{c,f\}\circ\texttt{pred} \equiv f\circ\{c,f\}$.



Par exemple, posons $X = \texttt{Int}$, $c = \texttt{zero}$, $f = \texttt{succ} \circ \texttt{succ}$ et $g = \texttt{pred} \circ \texttt{pred}$. On vérifie que $g \circ f \equiv \texttt{id}_{\texttt{Int}}$ et $f \circ g \equiv \texttt{id}_{\texttt{Int}}$. La règle fournit donc une nouvelle opération $\{c,f,g\} : \texttt{Int} \to \texttt{Int}$ telle que $\{c,f,g\} \circ \texttt{zero} \equiv \texttt{zero}$, $\{c,f,g\} \circ \texttt{succ} \equiv \texttt{succ} \circ \texttt{succ} \circ \{c,f,g\}$ et $\{c,f,g\} \circ \texttt{pred} \equiv \texttt{pred} \circ \texttt{pred} \circ \{c,f,g\}$. Dans le modèle M_Z, la fonction $\{c,f,g\}$ est interprétée par l'application $M(\{c,f,g\}) : \mathbb{Z} \to \mathbb{Z}$ telle que $n \mapsto 2n$.

3.5 Les types abstraits de données

La notion de *type abstrait de données* a été introduite dans les années 1970 afin de pouvoir manipuler les types de données (entiers, flottants, booléens, listes, etc) indépendamment de leur *implantation*. Cette approche introduisait de façon précise la distinction entre la spécification d'un programme (abstraite) et son implantation (concrète). Ces idées ont été utilisées par de nombreux langages, en particulier par Ada. La théorie des types abstraits de données repose sur le théorème 3.3, qui dit que toute spécification équationnelle S_1 a un modèle "canonique", qui est construit de façon "naturelle" à partir des termes de S_1. Ce théorème implique que, si deux termes de S_1 ont la même interprétation dans I, alors ont la même interprétation dans tout autre modèle M de S_1.

Définition 3.3 Etant donnée une spécification équationnelle S_1, le *type abstrait de données* déterminé par S_1 est la classe de tous les modèles initiaux de S_1.

Cette classe contient le modèle I de S_1, formé des termes clos modulo les congruences, et tous les autres modèles de S_1 qui sont isomorphes à I. Au niveau de la spécification, tous les modèles initiaux de S_1 jouent le même rôle. Par contre, lorsqu'on implante un type abstrait de données, on doit choisir un des modèles initiaux.

Par exemple le type abstrait de données "entier" peut être déterminé par la spécification $\mathtt{INT}_1$. Cela signifie qu'un entier peut être représenté par un terme clos de $\mathtt{INT}_1$, par exemple l'entier 1 par $\mathtt{succ} \circ \mathtt{zero} : \mathtt{Void} \to \mathtt{Int}$, et que deux termes représentent le même entier si et seulement si ils sont congrus, par exemple $\mathtt{succ} \circ \mathtt{zero}$ et $\mathtt{succ} \circ \mathtt{succ} \circ \mathtt{pred} \circ \mathtt{zero}$ représentent tous les deux l'entier 1. De ce point de vue, les entiers sont représentés par les mots sur l'alphabet $\{\mathtt{succ}, \mathtt{pred}, \mathtt{zero}\}$, avec un seul $\mathtt{zero}$ qui doit être le dernier caractère ; cet ensemble est souvent noté $(\mathtt{succ} + \mathtt{pred})^*.\mathtt{zero}$. Avec cette représentation, tester l'égalité de deux entiers n'est pas immédiat : par exemple, les termes $\mathtt{pred}(\mathtt{pred}(\mathtt{succ}(\mathtt{pred}(\mathtt{succ}(\mathtt{zero})))))$ et $\mathtt{succ}(\mathtt{succ}(\mathtt{pred}(\mathtt{pred}(\mathtt{pred}(\mathtt{zero})))))$ représentent le même entier...
Par ailleurs, le modèle M_Z de $\mathtt{INT}_1$ est isomorphe à I, c'est donc aussi un modèle initial de $\mathtt{INT}_1$. Par contre, le modèle M_2 "des entiers modulo 2" de $\mathtt{INT}_1$, analogue à celui qu'on a vu au chapitre 1, n'est pas isomorphe à I : l'ensemble $M_2(\mathtt{Int})$ a deux éléments alors que $I(\mathtt{Int})$, comme $M_Z(\mathtt{Int}) = \mathbb{Z}$, a une infinité d'éléments. Par conséquent, M_2 n'est pas un modèle de $\mathtt{INT}_2$.

3.6 La spécification, en pratique

La notion de spécification introduite dans les années 1970, fondée sur le théorème 3.3, continue d'être développée ; elle a donné lieu au langage de spécification CASL qui est pour le moment un outil de recherche.

Par ailleurs, la distinction entre spécification et implantation est maintenant intégrée, de façon plus ou moins nette, dans la plupart des langages, en particulier dans les langages orientés objets.

De plus, la spécification "dure" préconisée à l'origine, difficile à mettre en œuvre, a donné lieu à divers styles de spécification "molle", plus réalistes :
– la spécification *semi-formelle* de UML (Unified Modeling Language) est utilisée en milieu industriel pour gérer de gros projets logiciels, bien que (ou parce que ?) sa sémantique n'est pas très bien définie ;
– la spécification *partielle* de JML (Java Modeling Language), en cours de développement, permet d'imposer et de vérifier certaines propriétés des programmes en Java, lorsqu'on le juge utile ;
– la spécification *approchée* est encore au stade de la recherche, elle permet de construire une spécification par approximations successives.

3.7 Exercices

Utiliser la spécification $\mathtt{INT}_2$ pour spécifier l'application $n \mapsto 2n$, de $\mathbb{Z}$ vers $\mathbb{Z}$.

Utiliser la spécification $\mathtt{NAT}_2$ pour définir l'application de Fibonacci $Fib : \mathbb{N} \to \mathbb{N}$ telle que
$Fib(0) = 1$, $Fib(1) = 1$, et $Fib(n+2) = Fib(n+1) + Fib(n)$ pour tout $n \in \mathbb{N}$.
On peut considérer d'abord l'application $Fib' : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ telle que
$Fib'(n) = (Fib(n), Fib(n+1))$.

Décrire une spécification dont l'unique modèle est formé des listes de naturels.

Décrire une spécification des monoïdes, puis l'enrichir pour trouver $\{a, b\}^*$ comme modèle de termes.

4 Types de fonctions (21/10/03)

Depuis le chapitre 1, on utilise les types pour spécifier des ensembles et les fonctions pour spécifier des applications. Dans ce chapitre, on utilise les types de fonctions pour spécifier des ensembles d'applications.

4.1 Ensembles d'applications

Si on considère l'ensemble $\mathbb{R}$ des nombres réels, le sinus sin est une application de $\mathbb{R}$ dans $\mathbb{R}$:

$$\sin : \mathbb{R} \to \mathbb{R} : x \mapsto \sin(x) \ .$$

En effet, sin prend comme argument un nombre réel et retourne un nombre réel. Par exemple $\sin(\pi) = 0$.

Si on considère maintenant la dérivation, c'est aussi une application. Mais elle prend comme argument une application de $\mathbb{R}$ dans $\mathbb{R}$, et elle retourne comme résultat une application de $\mathbb{R}$ dans $\mathbb{R}$! Pour exprimer cela, on note :

$$\mathbb{R}^{\mathbb{R}}$$

l'ensemble des applications de $\mathbb{R}$ dans $\mathbb{R}$, par exemple $\sin \in \mathbb{R}^{\mathbb{R}}$. Alors la dérivation est une application de $\mathbb{R}^{\mathbb{R}}$ dans $\mathbb{R}^{\mathbb{R}}$:

$$\mathrm{diff} : \mathbb{R}^{\mathbb{R}} \to \mathbb{R}^{\mathbb{R}} : f \mapsto f' \ .$$

Par exemple $\mathrm{diff}(\sin) = \cos \in \mathbb{R}^{\mathbb{R}}$.

Plus généralement, étant donnés deux ensembles A et B, on note :

$$B^A = \{\varphi \mid \varphi : A \to B\}$$

l'ensemble des applications de A dans B. Par exemple, en notant $\mathbb{B} = \{v, f\}$ l'ensemble des booléens, $\mathbb{B}^A$ désigne l'ensemble des prédicats sur A.

Attention ! ici on utilise, pour une opération sur les ensembles, le même mot "puissance" et la même notation "$-^-$", que pour une opération sur les nombres. Il ne faut pas les confondre, mais quand même il y a un rapport entre les deux : si les ensembles A et B sont finis, alors l'ensemble B^A aussi, et $Card(B^A) = Card(B)^{Card(A)}$.

Dès qu'on a un élément φ de B^A et un élément x de A, on peut *appliquer* l'application φ à la valeur x, on obtient l'élément $y = \varphi(x)$ de B. Cela définit une application :

$$\texttt{appl} : B^A \times A \to B \ .$$

Considérons trois ensembles A, B et C et une application $\varphi : C \times A \to B$ à deux arguments et à valeurs dans B. Alors φ définit une application $\widehat{\varphi} : C \to B^A$ à un argument et à valeurs dans B^A :

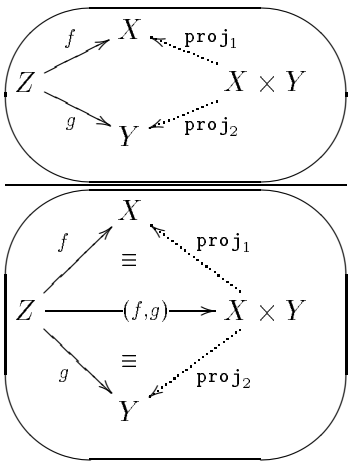
$$\varphi(c, a) = \widehat{\varphi}(c)(a) = \texttt{appl}(\widehat{\varphi}(c), a) \ .$$

En quelque sorte, l'application φ a deux arguments simultanés, alors que l'application $\widehat{\varphi}$ a deux arguments successifs. Cette application $\widehat{\varphi}$ s'appelle la *curryfiée* de φ. Le mot "curryfier" est formé d'après le nom du logicien Haskell Curry, par ailleurs son prénom a été utilisé pour nommer le langage de programmation Haskell...

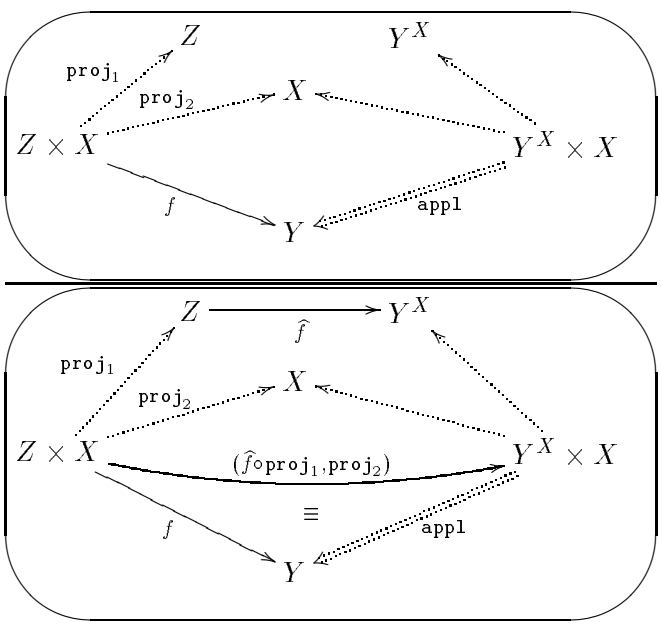
Remarque : le mot "application" est employé ici en deux sens différents. En anglais, on pourrait utiliser le mot "map" pour parler des applications en général, et alors on pourrait réserver le mot "application" pour les applications très particulières que sont les `appl`.

4.2 Types de fonctions

Rappel : au chapitre 1, pour introduire les paires, on a procédé de la façon suivante.
– Dès qu'on a deux types X et Y, on peut ajouter un nouveau type $X \times Y$, avec deux fonctions de projection $\mathtt{proj}_1 : X \times Y \to X$ et $\mathtt{proj}_2 : X \times Y \to Y$, on appelle cela le produit (potentiel) de X et Y, et on dessine les projections avec des flèches pointillées.
– Ensuite, dès qu'on a un type Z et deux fonctions $f : Z \to X$ et $g : Z \to X$, on peut ajouter une fonction $(f, g) : Z \to X \times Y$, avec les deux équations $\mathtt{proj}_1 \circ (f, g) \equiv f$ et $\mathtt{proj}_2 \circ (f, g) \equiv g$; cela peut être dessiné sous forme d'une règle :



Maintenant, pour introduire la curryfiée d'une fonction, on procède de façon similaire.
– Dès qu'on a deux types X et Y, on peut ajouter un nouveau type Y^X, avec une fonction d'application $\mathtt{appl} : Y^X \times X \to Y$, qu'on dessine avec une flèche "doublement pointillée".
– Ensuite, dès qu'on a un type Z et une fonction $f : Z \times X \to Y$, on peut ajouter une fonction $\widehat{f} : Z \to Y^X$, avec l'équation $\mathtt{appl} \circ (\widehat{f} \circ \mathtt{proj}_1, \mathtt{proj}_2) \equiv f$, où $\mathtt{proj}_1 : Y^X \times X \to Y^X$ et $\mathtt{proj}_2 : Y^X \times X \to X$ sont les projections ; cela peut être dessiné sous forme d'une règle :



Par exemple, considérons une spécification des naturels avec une fonction $\mathtt{ajout} : \mathtt{Nat} \times \mathtt{Nat} \to$

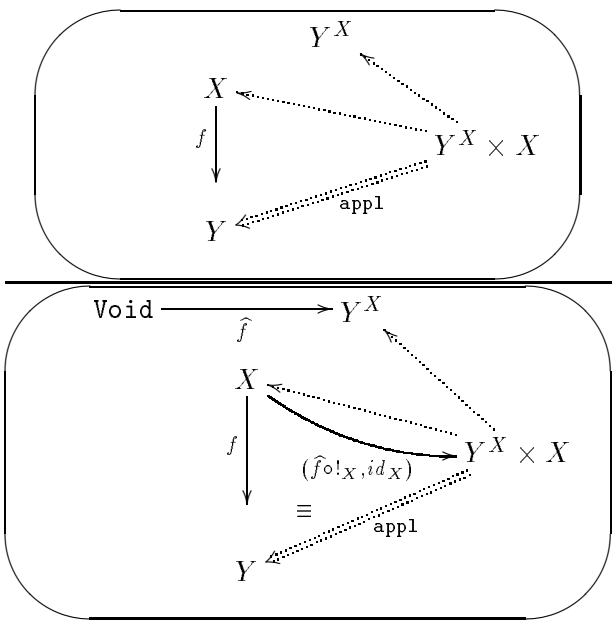
$\mathtt{Nat}$, alors on peut ajouter le type $\mathtt{Nat}^{\mathtt{Nat}}$ et la fonction $\widehat{\mathtt{ajout}} : \mathtt{Nat} \to \mathtt{Nat}^{\mathtt{Nat}}$. Dans le modèle "des naturels", ces deux fonctions sont interprétées respectivement par l'addition :

$$(x, y) \mapsto x + y$$

et par l'addition curryfiée :

$$x \mapsto (y \mapsto x + y) \ .$$

Dans le cas particulier où $Z = \mathtt{Void}$, on peut identifier $\mathtt{Void} \times X$ et X, les projections deviennent alors $!_X : X \to \mathtt{Void}$ et $id_X : X \to X$, et la règle devient :
– Dès qu'on a une fonction $f : Z \times X \to Y$, on peut ajouter une fonction constante $\widehat{f} : \mathtt{Void} \to Y^X$, avec l'équation $\mathtt{appl} \circ (\widehat{f} \circ !_X, id_X) \equiv f$.
Le dessin est alors plus simple, mais moins symétrique :



Par exemple, considérons une spécification des naturels avec une fonction $\mathtt{succ} : \mathtt{Nat} \to \mathtt{Nat}$, alors on peut ajouter le type $\mathtt{Nat}^{\mathtt{Nat}}$ et la fonction $\widehat{\mathtt{succ}} : \mathtt{Void} \to \mathtt{Nat}^{\mathtt{Nat}}$. Dans le modèle "des naturels", ces deux fonctions sont interprétées respectivement par l'application "successeur" :

$$x \mapsto x + 1$$

et par l'application constante "successeur curryfiée" :

$$* \mapsto (x \mapsto x + 1) \ .$$

4.3 Un peu de lambda-calcul

Le *lambda-calcul pur* a été introduit par Alonzo Church dans les années 1930. Ce calcul construit des fonctions et les applique à des arguments. Les trois ingrédients essentiels du lambda-calcul pur sont l'abstraction, l'application et la beta-réduction. Les termes du lambda-calcul pur sont formés inductivement, à partir d'un ensemble (infini dénombrable) de variables :
– toute variable x est un terme,

– l'*abstraction* permet de former un terme $\lambda\, x\, .\, t$ à partir de toute variable x et tout terme t,
– l'*application* permet de former un terme $t_1(t_2)$ à partir de tous termes t_1 et t_2.
Ce calcul possède une seule règle :
– la *beta-réduction* consiste à remplacer $(\lambda\, x\, .\, t)(u)$ par $t[x := u]$, ce qui désigne le terme t dans lequel la variable x est remplacée par le terme u :

$$(\lambda\, x\, .\, t)(u) \rightsquigarrow_\beta t[x := u]\ .$$

L'idée est que $\lambda\, x\, .\, t$ représente la fonction qui à la variable x associe t, et que $t_1(t_2)$ représente l'application de la fonction t_1 à la valeur t_2. Cependant, il n'y a pas de distinction entre fonction et valeur, de sorte qu'on peut former des termes comme $\lambda\, x\, .(x(x))$, qui semble "incohérent".
Un calcul consiste alors à partir d'un terme quelconque et à (tenter de) le *normaliser* en appliquant des β-réductions. Un terme est en *forme normale* si on ne peut plus lui appliquer aucune β-réduction, c'est-à-dire s'il ne contient aucun sous-terme de la forme $(\lambda\, x\, .\, t)(u)$.
Mais il peut arriver que la réduction d'un terme ne termine pas, par exemple $v(v)$ où $v = \lambda\, x(x(x))$: en effet ce terme est de la forme $(\lambda\, x\, .\, t)(u)$ où $t = x(x)$ et $u = v$, donc il se réduit en $t[x := u]$, qui vaut $v(v)$, or c'est la valeur de départ :

$$(\lambda\, x(x(x)))\ (\lambda\, x(x(x))) \rightsquigarrow_\beta (\lambda\, x(x(x)))\ (\lambda\, x(x(x)))\ .$$

Le lambda-calcul pur a donné lieu aux langages Lisp, Scheme, etc.

Cependant, le lambda-calcul pur amène à des paradoxes similaires à ceux de la théorie des ensembles (paradoxe de Russell). Une façon de résoudre ces paradoxes est, comme pour les ensembles, d'introduire une notion de type.

Dans le *lambda-calcul (simplement) typé*, au même titre que les types de base `Nat`, `Int`, `Bool`, etc, il y a les types de fonctions $\texttt{Nat}^{\texttt{Nat}}$, $\texttt{Bool}^{\texttt{Int}}$, $\texttt{Bool}^{\texttt{Nat}^{\texttt{Nat}}}$, etc. On note parfois aussi $X \to Y$ le type Y^X. Les trois ingrédients essentiels du lambda-calcul typé sont encore l'abstraction, l'application et la beta-réduction, mais leur utilisation est restreinte par le typage.
Les types sont eux-mêmes construits inductivement, à partir d'un ensemble (fini) de symboles de types appelés types de base :
– tout type de base est un type,
– pour tous types X et Y, il y a un type Y^X.
Ensuite les termes du lambda-calcul typé, avec leur type, sont formés inductivement à partir d'un ensemble (infini dénombrable) de variables typées (on note $t : X$ le fait que le terme t est de type X) :
– toute variable x de type X est un terme $x : X$,
– l'*abstraction* permet de former un terme $\lambda\, x\, .\, t : Y^X$ à partir de toute variable $x : X$ et tout terme $t : Y$,
– l'*application* permet de former un terme $t_1(t_2) : Y$ à partir de tous termes $t_1 : Y^X$ et $t_2 : X$; les types introduisent ici une restriction importante.
Ce calcul possède aussi la règle de beta-réduction, sous forme typée :
– la *beta-réduction* consiste à remplacer $(\lambda\, x\, .\, t)(u)$ par $t[x := u]$, à condition que le terme $(\lambda\, x\, .\, t)(u)$ soit bien typé, c'est-à-dire que $x : X$, $t : Y$ et $u : X$; ici aussi, les types introduisent une restriction importante.
Un résultat important du lambda-calcul typé est le *théorème de normalisation forte* :

Théorème 4.1 *Il ne peut pas y avoir de réduction infinie d'un terme dans le lambda-calcul typé.*

Le lambda-calcul typé a donné lieu aux langages ML, Caml, Haskell, etc. qui implantent des statégies différentes pour la réduction.

Dans ce qu'on a vu précédemment, la règle pour la construction de la curryfiée, lorsque $Z = \texttt{Void}$, fournit les trois ingrédients du lambda-calcul typé :
– l'*abstraction* correspond à la curryfication, qui associe à toute fonction $f : X \to Y$ une constante $\widehat{f} : \texttt{Void} \to Y^X$,
– l'*application* est la fonction $\texttt{appl} : Y^X \times X \to Y$,
– la *beta-réduction* consiste à utiliser l'équation $\texttt{appl} \circ (\widehat{f} \circ !_X, id_X) \equiv f$ de gauche à droite.

4.4 Un aperçu de la théorie des types

On peut maintenant donner une idée de ce qui s'appelle la *théorie des types* en informatique. Dans cette théorie, on établit une correspondance, appelée *l'isomorphisme de* CURRY-HOWARD, qui permet d'interpréter des termes informatiques comme des formules logiques. On donne ici quelques indications sur cette correspondance, à partir d'une spécification S comportant deux types X et Y, et d'un modèle de S qui interprète X comme l'ensemble $[p]$ des preuves d'une proposition p, et Y comme l'ensemble $[q]$ des preuves d'une proposition q.

Implication.

Si S comporte le type puissance $Y^X = X \to Y$, alors celui-ci est interprété comme l'ensemble $[q]^{[p]} = [p] \to [q]$ des applications de $[p]$ vers $[q]$. Or toute application de $[p]$ vers $[q]$ associe à toute preuve de p une preuve de q, autrement dit elle fournit une preuve de $p \Rightarrow q$. Réciproquement, toute preuve de $p \Rightarrow q$ associe à toute preuve de p une preuve de q, autrement dit c'est une application de $[p]$ vers $[q]$. Par conséquent, l'ensemble $[p] \to [q]$ des applications de $[p]$ dans $[q]$ est l'ensemble $[p \Rightarrow q]$ des preuves de $p \Rightarrow q$:

$$[p] \to [q] = [p \Rightarrow q] \ .$$

C'est le premier pas vers l'isomorphisme de Curry-Howard : si le type X correspond à l'ensemble des preuves de p et le type Y à l'ensemble des preuves de q, alors le type Y^X correspond à l'ensemble des preuves de $p \Rightarrow q$.

Conjonction.

Si S comporte le type produit $X \times Y$, alors celui-ci est interprété comme l'ensemble $[p] \times [q]$ des couples formés d'une preuve de $[p]$ et d'une preuve de $[q]$. Or si on a une preuve de p et une preuve de q, on a bien sûr une preuve de $p \wedge q$. Réciproquement, une preuve de $p \wedge q$ fournit une preuve de p et une preuve de q. Par conséquent, l'ensemble $[p] \times [q]$ des couples formés d'une preuve de $[p]$ et d'une preuve de $[q]$ est l'ensemble $[p \wedge q]$ des preuves de $p \wedge q$:

$$[p] \times [q] = [p \wedge q] \ .$$

C'est un second pas vers l'isomorphisme de Curry-Howard : si le type X correspond à l'ensemble des preuves de p et le type Y à l'ensemble des preuves de q, alors le type $X \times Y$ correspond à l'ensemble des preuves de $p \wedge q$.

Disjonction.

Si S comporte le type somme $X + Y$, alors celui-ci est interprété comme l'ensemble $[p] + [q]$ réunion disjointe des preuves de $[p]$ et des preuves de $[q]$. Or si on a une preuve de p ou une preuve de q, on a bien sûr une preuve de $p \vee q$. Mais réciproquement, une preuve de $p \vee q$ ne

fournit pas forcément une preuve de p ou une preuve de q. Par exemple, lorsque $q = \neg p$, la proposition $p \vee \neg p$ est toujours vraie, même lorsque p est une conjecture difficile dont personne ne sait si elle est vraie ou fausse. Par conséquent, l'ensemble $[q] + [p]$ réunion disjointe des preuves de $[p]$ et des preuves de $[q]$ **n'est pas** l'ensemble $[p \vee q]$ des preuves de $p \vee q$:

$$[p] + [q] \neq [p \vee q] \text{ en logique classique.}$$

Cependant, si on remplace la logique classique par la *logique intuitioniste*, alors il devient vrai que :

$$[p] + [q] = [p \vee q] \text{ en logique intuitioniste.}$$

Pour aller plus loin dans la description de l'isomorphisme de Curry-Howard, il faudrait donc commencer par définir la logique intuitioniste, ce qui ne sera pas fait dans ce cours.

4.5 Exercice

Utiliser la spécification $\mathtt{NAT}_2$ du chapitre 3 pour spécifier par récurrence l'addition sur les naturels. L'idée est de définir par récurrence la fonction currifiée $\widehat{\mathtt{ajout}} : \mathtt{Nat} \to \mathtt{Nat}^{\mathtt{Nat}}$, plutôt que la fonction à deux arguments $\mathtt{ajout} : \mathtt{Nat} \times \mathtt{Nat} \to \mathtt{Nat}$.

5 Types et classes

On aborde ici les notions de spécification et de type dans les langages orientés objet, plus précisément dans le langage Java. Un nouvel élément apparaît, qui est commun aux langages impératifs et aux langages orientés objets : les fonctions peuvent utiliser et modifier un *état*, qui n'est pas explicitement mentionné dans la syntaxe du langage.

5.1 Classes, interfaces et types en Java

En Java, il existe deux catégories de types : les types primitifs et les types composés.

Les *types primitifs* ne sont pas construits à partir d'autres types. Ce sont : `int`, `float`, `double`, `char`, `boolean`, etc.

Les *types composés* sont (essentiellement) les types associés aux classes et aux interfaces, ils sont construits à partir d'autres types. On note de la même manière une classe (ou une interface) et son type.

Java est un langage *fortement typé* : dès sa déclaration, toute variable a un type déterminé. Une variable dont le type est un type primitif est un emplacement-mémoire qui contient sa valeur. Une variable dont le type est une classe fera référence à un objet de cette classe. Une variable dont le type est une interface fera référence à un objet d'une classe qui implante cette interface.

Une interface comporte seulement une *spécification* (partie abstraite), alors qu'une classe comporte à la fois une *spécification* et une *implantation* (partie concrète), cependant le type d'une classe ne dépend que de sa partie abstraite.
Le *profil* d'une méthode est formé des types de ses arguments et du type de son résultat.
Une interface est formée (essentiellement) du nom de l'interface et des noms et profils de ses méthodes. On ne peut pas instancier une interface `I` (pas de `new()`), par contre puisqu'elle définit un type, on peut déclarer des variables de ce type ; et on peut affecter à un variable déclarée de type `I` tout objet d'une classe qui implante `I`.
La spécification d'une classe est formée (essentiellement) du nom de la classe et des noms et profils de ses méthodes. Par ailleurs, la partie implantation de la classe comporte (essentiellement) ses variables d'instance et les corps pour ses méthodes.

Récapitulons :
– une interface définit un (et un seul) type,
– une classe définit un (et un seul) type,
– un objet est engendré par une (et une seule) classe,
– donc un objet a un (et un seul) type.

5.2 Un exemple

Dans les exemples traités ici, pour bien distinguer la spécification de l'implantation, on restreint l'utilisation des classes en Java : une classe implante une interface et n'a pas de méthode en propre (sauf d'éventuelles méthodes privées). Donc une classe `C` ne fait que coder les méthodes de l'interface `I` que `C` implante. Avec ces restrictions, on peut éviter de parler des types définis par les classes, et se limiter aux types définis par les interfaces : on a d'une part la *spécification*

(par les *interfaces*) et l'*implantation* (par les *classes*) :

spécification	interface
implantation	classe(s)

Par exemple, considérons une interface `VAR` :

```
interface VAR {
// methodes : //
    int get();
    void set(int n);
    void incr();
}
```

et deux classes distinctes $\texttt{VAR}_1$ et $\texttt{VAR}_2$ qui implantent l'interface `VAR` :

```
class VAR1 implements VAR {
// variable d'instance : //
   int x;
// constructeur : //
   VAR1() {x=0;}
// definition des methodes : //
   public int get() {return x;}
   public void set(int n) {x=n;}
   public void incr() {set(get()+1);}
}
```

L'exécution de :

```
VAR1 x1 = new VAR1();
System.out.println("valeur x1 : "+x1.get());
x1.set(3);
System.out.println("valeur x1 : "+x1.get());
x1.incr();
System.out.println("valeur x1 : "+x1.get());
```

imprime les valeurs 0, 3 et 4.

```
class VAR2 implements VAR {
// variable d'instance : //
   int x;
// constructeur : //
   VAR2() {x=-1;}
// definition des methodes : //
   public int get() {return x+2;}
   public void set(int n) {x=n-1;}
   public void incr() {set(get()+3);}
}
```

L'exécution de :

```
VAR2 x2 = new VAR2();
System.out.println("valeur x2 : "+x2.get());
x2.set(3);
System.out.println("valeur x2 : "+x2.get());
x2.incr();
System.out.println("valeur x2 : "+x2.get());
```

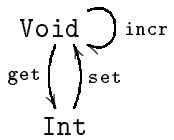
imprime les valeurs 1, 7 et 11.

5.3 En Java, un type détermine deux signatures

Rappelons qu'une *signature* est une spécification équationnelle sans équation. Par conséquent, une interface `I` en Java définit une signature $\texttt{IMPL}_0(\texttt{I})$: il suffit pour obtenir $\texttt{IMPL}_0(\texttt{I})$ de passer de la notation Java "`B m(A a)`" à la notation des chapitres précédents "$m : A \to B$".

Par exemple, le type entier étant spécifié par une signature $\texttt{INT}_0$ comportant le type `Int`, l'interface `VAR` correspond à la signature :

Signature $\texttt{IMPL}_0(\texttt{VAR})$:
Enrichit : $\texttt{INT}_0$;
Fonctions : `get` : `Void` $\to$ `Int`, `set` : `Int` $\to$ `Void`, `incr` : `Void` $\to$ `Void`.



Comme toute spécification, cette signature a des modèles. Mais ceux-ci ne sont pas intéressants, car ils ne font pas intervenir l'*état* de l'objet. Par exemple, puisque la seule application de $\{*\}$ vers $\{*\}$ est l'identité $* \mapsto *$, la fonction `incr` : `Void` $\to$ `Void` est forcément interprétée comme l'application identité, qui ne fait rien, alors qu'on aimerait que l'interprétation de `incr` incrémente la valeur de la variable.

On va voir maintenant comment construire, à partir d'une interface `I`, une autre signature $\texttt{EXPL}_0(\texttt{I})$, qui correspond beaucoup mieux à la sémantique visée. C'est un procédé simple, qu'on appelle ici un *zoom*. Il permet de passer :
– de la signature $\texttt{IMPL}_0(\texttt{I})$, qui est une traduction directe de `I`, où l'état est *implicite*,
– à la signature $\texttt{EXPL}_0(\texttt{I})$, où l'état est *explicite*.

Le zoom ajoute un type et modifie les fonctions.
– On ajoute à la signature un nouveau type : c'est le type `I` qui est associé à l'interface `I`, et qui représente l'ensemble des états possibles pour chaque objet de type `I`.
– On modifie les profils des méthodes pour tenir compte du fait que la plupart des méthodes peuvent utiliser et modifier l'état de l'objet. Autrement dit, en général une méthode $f : A \to B$ est remplacée par :

$$f : A \times \texttt{I} \to B \times \texttt{I} \ ,$$

y compris lorsque A ou B est `Void`, en posant $\texttt{Void} \times \texttt{I} = \texttt{I}$, ce qui est justifié puisqu'on peut identifier les ensembles $\{*\} \times X$ et X, par la bijection $(*, x) \leftrightarrow x$.
– Cependant, on procède différemment lorsque f est une méthode *pure* ; cela signifie qu'elle ne provoque pas d'effet de bord, c'est-à-dire qu'elle ne modifie pas l'état. Une méthode pure $f : A \to B$ est remplacée par :

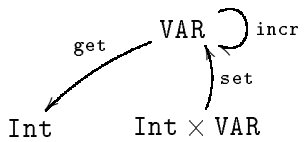
$$f : A \times \texttt{I} \to B \ .$$

– Enfin, les opérateurs sur les types de données sont complètement indépendants de l'état ; un tel opérateur $f : A \to B$ reste :

$$f : A \to B \ .$$

Par exemple, pour l'interface `VAR`, la méthode `get` est pure, mais pas `set` ni `incr`. On obtient donc, après le zoom, la signature :

Signature $\texttt{EXPL}_0(VAR)$:
Enrichit : $\texttt{INT}_0$;
Type : `VAR`;
Fonctions : $\texttt{get} : \texttt{VAR} \to \texttt{Int}$, $\texttt{set} : \texttt{Int} \times \texttt{VAR} \to \texttt{VAR}$, $\texttt{incr} : \texttt{VAR} \to \texttt{VAR}$.



SYNTAXE avec état implicite	$f : X \to Y$ données	$f : X \to Y$ pure	$f : X \to Y$ générale
SYNTAXE avec état explicite	$f : X \to Y$	$f : X \times \texttt{I} \to Y$	$f : X \times \texttt{I} \to Y \times \texttt{I}$

5.4 Avec JML, un type détermine deux spécifications

Un type en Java ne permet pas de parler des propriétés des méthodes, mais seulement de leur profil. Pour exprimer certaines propriétés des méthodes, on peut utiliser le langage JML (Java Modeling Language). Cette démarche est similaire à celle qui consiste, au chapitre 1, à ajouter des équations à une signature pour former une spécification équationnelle.

JML est un *langage de spécification* qui peut être utilisé pour décrire le comportement des classes et des interfaces de Java. Ce langage permet d'ajouter des éléments de spécification, appelés *assertions*, à différents niveaux dans les programmes Java :
– des *invariants* pour les classes et les interfaces,
– des *pré-conditions* et *post-conditions* pour les méthodes,
– et diverses assertions au milieu du code Java.

Le compilateur JML (`jmlc`) permet de compiler des interfaces et des classes Java accompagnées de spécifications JML en bytecode Java. Ce compilateur produit des fichiers en bytecode qui vérifient à l'exécution les assertions JML.

Une assertion JML se présente comme un commentaire particulier :

```
//@ Une assertion JML sur une ligne.

/*@ Début d'une assertion JML...
  @ ...suite...
  @ ...fin de l'assertion.
  @*/
```

`/*@ pure @*/`.
Les conditions qui figurent dans les assertions JML sont écrites en Java, sous forme d'expressions booléennes. Mais ces expressions doivent être *purement fonctionnelles*, c'est-à-dire sans effet de bord, sinon l'état serait modifié lors de la vérification des assertions JML, à l'exécution. Cela signifie que seules les méthodes pures peuvent être utilisées dans les assertions JML. Pour spécifier qu'une méthode est pure, et que par conséquent on peut l'utiliser dans les assertions JML, on utilise (juste avant la déclaration de la méthode) l'assertion `/*@ pure @*/`.

`assert`.
Le mot-clé `assert` est utilisé pour placer une assertion dans le corps d'une méthode. Elle fait référence à l'état *à ce point du programme*.

Par ailleurs, un état *visible* est un état du système avant ou après l'exécution d'une méthode (et pas pendant l'exécution). Toutes les assertions JML, sauf `assert`, portent sur des états visibles.

`invariant`.
Le mot-clé `invariant` précède un *invariant* de classe ou d'interface, c'est-à-dire une condition qui doit être vérifiée dans tout état visible.

`requires`.
Une *pré-condition* pour une méthode m est introduite par `requires`. C'est une condition portant sur :
– l'état de l'objet o qui invoque la méthode m,
– les paramètres de la méthode m,
et qui doit être satisfaite avant chaque appel de m.

`ensures`.
Une *post-condition* pour une méthode m est introduite par : `ensures`. C'est une condition portant sur :
– l'état de l'objet o avant l'exécution de la méthode,
– l'état de l'objet o après l'exécution de la méthode,
– les paramètres de la méthode m,
– le résultat de la méthode m,
et qui doit être satisfaite après chaque appel de m.

Lorsqu'une méthode est susceptible de lever des exceptions, une post-condition ne concerne que ce qui se passe en cas de terminaison normale, il existe un autre mot-clé pour les post-conditions exceptionnelles.

`\old(..)` et `\result`.
Dans les post-conditions, et *uniquement* dans les post-conditions, on dispose de l'opérateur `\old(..)` et de la variable `\result`. La valeur de l'opérateur `\old(..)` est calculée immédiatement avant l'exécution du code de la méthode. La valeur de la variable `\result` est la valeur de l'expression retournée par la méthode, lorsque le type de retour de la méthode est différent de `Void`.

De plus, on peut aussi utiliser \forall et \exists.

Reprenons l'exemple de l'interface VAR :

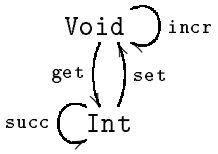
```
/**
 * Interface pour une variable de type entier.
 */
interface VAR {
// methodes : //
    /**
     * Retourne la valeur de la variable.
     */
    /*@ pure @*/
    int get();
    /**
     * Affecte l'entier n comme valeur pour la variable.
     */
    /*@
      @ ensures
      @   get() == n;
      @*/
    void set(int n);
    /**
     * Incrémente la valeur de la variable.
     */
    /*@
      @ ensures
      @   get() == \old(get())+1;
      @*/
    void incr();
}
```

Avec cette spécification, l'implantation $\texttt{VAR}_1$ convient, mais pas l'implantation $\texttt{VAR}_2$. Le compilateur jmlc compile bien l'interface et les deux classes, mais il renvoie une erreur à l'exécution lorsqu'on utilise la classe $\texttt{VAR}_2$.

En revenant aux notations des chapitres précédents, le type entier étant défini par une spécification INT comportant le type Int et la fonction $\texttt{succ} : \texttt{Int} \to \texttt{Int}$, l'interface VAR avec ses assertions JML correspond à deux spécifications.

D'abord une spécification avec état implicite :

Spécification IMPL(VAR) :
Enrichit : INT;
Fonctions : $\texttt{get} : \texttt{Void} \to \texttt{Int}$ (pure), $\texttt{set} : \texttt{Int} \to \texttt{Void}$, $\texttt{incr} : \texttt{Void} \to \texttt{Void}$;
Equations : $\texttt{get} \circ \texttt{set} \equiv \texttt{id}_{\texttt{Int}}$, $\texttt{get} \circ \texttt{incr} \equiv \texttt{succ} \circ \texttt{get}$.

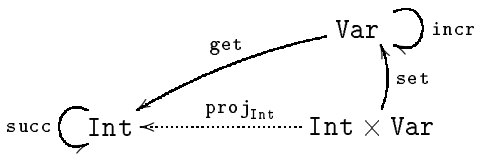


$\texttt{get} \circ \texttt{set} \equiv \texttt{id}_{\texttt{Int}}$

$\texttt{get} \circ \texttt{incr} \equiv \texttt{succ} \circ \texttt{get}$

Puis une spécification avec état explicite, dans laquelle on voit que les post-conditions correspondent à des équations *d'observation* : elles portent sur les données obtenues comme résultat des méthodes, mais pas directement sur les états.

Spécification `EXPL(VAR)` :
Enrichit : `INT`;
Fonctions : $\texttt{get} : \texttt{Var} \to \texttt{Int}$, $\texttt{set} : \texttt{Int} \times \texttt{Var} \to \texttt{Var}$, $\texttt{incr} : \texttt{Var} \to \texttt{Var}$;
Equations : $\texttt{get} \circ \texttt{set} \equiv \texttt{proj}_{\texttt{Int}}$, $\texttt{get} \circ \texttt{incr} \equiv \texttt{succ} \circ \texttt{get}$.



$\texttt{get} \circ \texttt{set} \equiv \texttt{proj}_{\texttt{Int}}$

$\texttt{get} \circ \texttt{incr} \equiv \texttt{succ} \circ \texttt{get}$

5.5 Héritage

En Java, une interface peut hériter d'une ou de plusieurs autres interfaces, et une classe peut hériter d'une autre classe, mais pas de plusieurs autres classes : l'héritage multiple est autorisé pour les interfaces, mais pas pour les classes. Les interfaces forment une hiérarchie, les classes forment une autre hiérarchie, et ces deux hiérarchies sont reliées par la relation d'implantation : une classe peut implanter une ou plusieurs interfaces.

Etant donnée une interface `I`, une interface $\texttt{I}'$ hérite de `I` si elle lui ajoute des méthodes. Le type $\texttt{I}'$ est appelé un *sous-type* du type `I`.

Etant donnée une classe `C`, une classe $\texttt{C}'$ hérite de `C` si elle lui ajoute des méthodes, ou des variables d'instance, ou si elle modifie l'initialisation de variables d'instance ou le corps de méthodes. La classe $\texttt{C}'$ est appelée une *sous-classe* de la classe `C`. En Java, on ne peut pas dans une sous-classe modifier les types des attributs ni les profils des méthodes.

Clairement, le type d'une sous-classe est un sous-type de la classe.

En Java, la redéfinition d'une méthode dans une sous-classe peut modifier complètement ses propriétés. Si on veut l'éviter, on peut utiliser JML, comme on l'a vu dans l'exemple.

5.6 Exercice

Une banque souhaite informatiser la gestion de ses comptes en utilisant Java. Elle souhaite que pour chaque compte on puisse lire le solde du compte et effectuer un dépôt.
-1- Traduire ce cahier des charges en une interface Java, nommée `Compte`, sachant que tous les nombres manipulés sont des entiers.
Dessiner la signature avec état implicite et la signature avec état explicite qui correspondent à l'interface `Compte`.

Trois sociétés informatiques sont contactées. La première propose une réalisation dans laquelle chaque compte est représenté par son solde. La seconde propose une réalisation dans laquelle

chaque compte est représenté par la liste de tous les soldes successifs. La troisième propose une réalisation dans laquelle chaque compte est représenté par la liste de tous les dépôts qui ont été effectués.
-2- Implanter les trois classes Java, nommées `Compte1`, `Compte2` et `Compte3`, correspondant à ces réalisations.

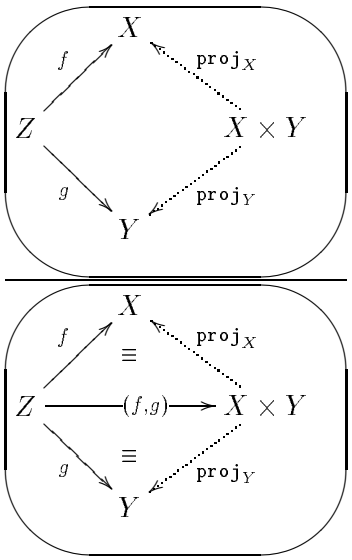
La banque souhaite ajouter deux opérations pour chaque compte : retrait et virement vers un autre compte.
-3- Enrichir l'interface et les trois classes pour réaliser ce nouveau cahier des charges.
Enrichir aussi les dessins des signatures.
Peut-on faire un virement d'un compte instance de `Compte2` vers un compte instance de `Compte3` ?

La banque craint de ne pas avoir été assez précise sur les opérations souhaitées. En fait, le solde d'un compte doit toujours être positif, les opérations de dépôt, retrait et virement ne peuvent concerner que des entiers positifs, et le solde doit varier après chacune de ces opérations conformément à ce que l'on attend.
-4- Ajouter à ce qui précède les assertions JML qui traduisent ces propriétés.
Peut-on faire un virement d'un compte vers lui-même ?

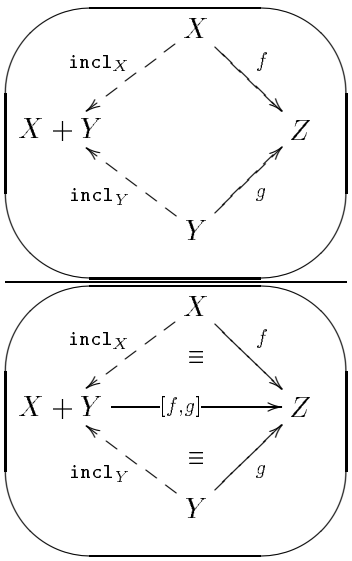
6 Annexe

On récapitule ici les principales règles introduites aux chapitres 1, 2, 3 et 4.

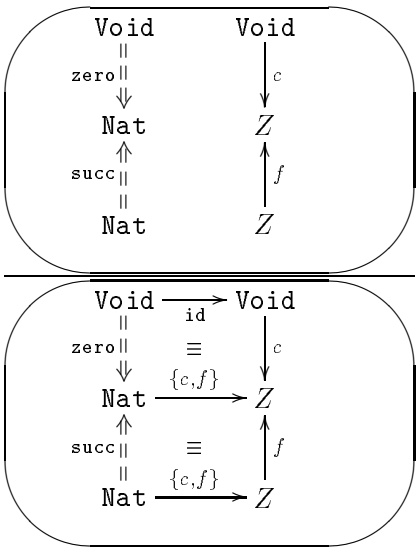
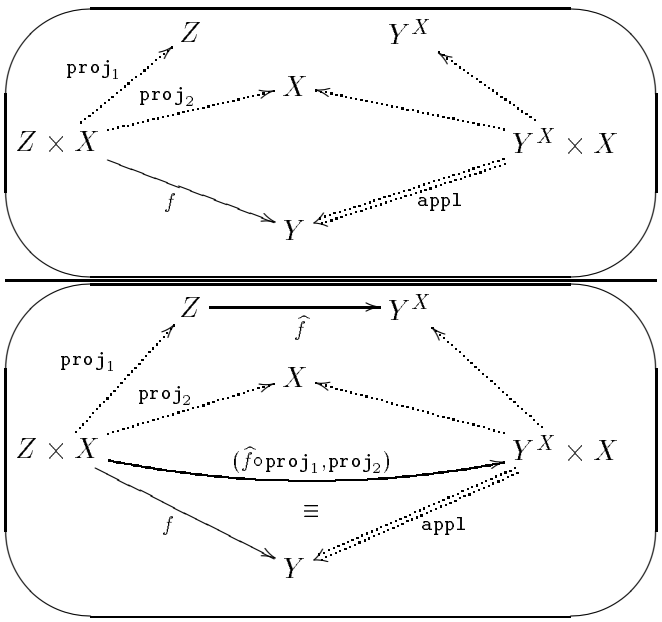
Construction de paires (chapitre 1).



Branchements (chapitre 2).



Récurrence (chapitre 3).



Currification (chapitre 4).