Etude du typage dans le sytème de calcul scientifique Aldor[1]

Emmanuel Touratier

Université de Limoges

[1]A Language for Describing Objects and their Relationships

Table des matières

Introduction

Apparu vers le milieu des années 70, le langage de calcul formel Axiom [13], initialement Scratchpad, faisait montre de caractéristiques peu communes dans l'univers des langages de calcul formel, comparativement aux langages contemporains ou ultérieurs tels Reduce, Macsyma, Mathematica, Maple, Magma, Mupad.

Aldor est le successeur direct du langage Axiom. Comme tous les langages de calcul formel, il autorise la manipulation des objets de base du calcul mathématique : entiers, flottants, booléens, chaînes de caractères, etc, en programmation impérative ou fonctionnelle.

`n : Integer := 2` est une affectation de variable

`n : Integer == 2` est une définition de constante

`carre(n : Integer) : Integer == n * n`
est une définition de fonction

`carre(2)` est une application de fonction

On dispose également en Aldor des types de "l'algèbre" : `Monoid, Ring, Field`, etc, et

```
R : Ring == Integer ;  R : Field == Float
```

sont des expressions licites dans le langage.

On définit de même des fonctions :

```
Polynome(R : Ring) :  Ring == ...
```

ce qui permet de construire le nouveau type `Polynome(Integer)`.

Si l'on peut dire que 2 est de type `Integer` ou que $x^2 + x + 1$ est de type `Polynome(Integer)`, on peut aussi dire en Aldor que `Integer` et `Polynome(Integer)` ne sont pas des types atomiques et qu'en plus ils ont eux-mêmes un type. Il n'y a pas de différence fondamentale entre 2, `Integer` et `Polynome(Integer)`.

Le trait le plus atypique de ce langage réside donc dans son système de types, d'une très grande richesse expressive. Il hérite des travaux menés sur les spécifications algébriques [9] dont il reprend les concepts.

Notre étude portera donc sur le système de types d'Aldor [19]. On abordera l'aspect standard du typage d'Aldor en tant que langage fonctionnel, avec les notions de typage par déclaration ou par inférence, de constructeur de types de fonctions $\rightarrow$ ou de types de uplets $(\ldots)$, etc. On verra que pour expliquer le typage de certaines expressions du langage, appelées domaines et catégories, on a besoin de l'approche du typage par les "types abstraits algébriques". Ceci nous permettra de justifier d'une part l'existence de catégories pour typer les domaines et d'autre part le fait que les catégories typent d'autres catégories, par une relation de satisfaction entre types que l'on définira.

Pour cette étude du système de types, on préfèrera aux spécifications algébriques la théorie des esquisses. Développée depuis 1968 par Ehresmann [8], la théorie des esquisses peut s'utiliser pour formaliser et étudier aussi bien la logique classique [10, 11] que des problèmes plus circonscrits de l'informatique [2]. A la lumière de travaux précédents [6, 7, 17], les esquisses et leurs généralisations se sont révélées être un bon formalisme pour l'étude des langages informatiques. L'objectif poursuivi dans ce travail est unique :

déterminer une esquisse associée au langage de calcul formel Aldor.

Celle-ci devra traduire la syntaxe du langage ainsi que toutes les caractéristiques de son système de types : construction, relation de satisfaction dont l'héritage n'est qu'un cas particulier, etc.

Dans une première partie, on traitera de manière générale du typage dans les langages informatiques. Puis on mettra en évidence l'originalité du typage d'Aldor tout en essayant de donner les raisons pour lesquelles ce choix

aurait été opéré par les concepteurs du langage. Ceci sera fait de manière informelle, repris par la suite à l'aide des procédés de construction de la théorie des esquisses.

Après avoir rappelé les notions essentielles de la théorie des esquisses, on pourra débuter la construction de l'esquisse associée à Aldor proprement dite. Elle repose sur la structure de catégorie cartésienne fermée, base de tous les langages fonctionnels.

Une fois posée la structure de base du langage, on définira précisément ce qu'est le type d'une expression Aldor, selon sa nature et sa présentation. Il sera ensuite possible de donner toutes les règles de satisfaction de types, relation qui nous permettra de justifier tous les mécanismes du typage d'Aldor. On verra en conclusion que la notion de typage est insuffisante pour exprimer la validité d'une expression dans le langage Aldor et qu'il faut y ajouter la notion de nature.

On pourra consulter ce document totalement ou partiellement selon ses centres d'intérêt en se référant au plan suivant :

- Chapitres 1 et 2 : le typage en général et celui d'Aldor en particulier
- Chapitre 3 : rappels sur la théorie des esquisses
- Chapitres 4, 5 et 6 : construction de l'esquisse depuis l'application de fonction jusqu'à l'inférence de types
- Chapitre 7 : retour sur quelques propriétés du typage à la lumière du point précédent

Chapitre 1

Typage

Le typage dans les langages de programmation a été introduit à l'origine pour répondre à des besoins de vérification dans la formation des termes: il fallait au moins s'assurer que le profil des opérateurs était respecté.

Mais le typage aide aussi à la compréhension des programmes. Un langage de programmation est en effet un outil qui permet de produire un programme en langage machine (ce qu'on appelle également le "code-objet") à partir d'un programme "code source", assorti d'un ensemble de commentaires écrits par et pour le programmeur. Lorsque ces commentaires ne sont pas traités par la machine, on parle de langage de bas niveau et dans ce cas la compilation d'un programme source consiste à étudier la partie écrite en code et à "oublier" les commentaires.

Dans un langage dit de haut niveau, une partie des commentaires peut être vérifiée par la machine et c'est précisément cette partie que l'on désigne par le terme typage.

Dans cette classe de langages de haut niveau, on peut encore distinguer les langages selon la richesse de leur système de types.

1.1 Les langages fonctionnels

Pour la plupart des langages fonctionnels, comme CAML par exemple, un **système de types** est une classe de formules d'un certain langage, ces formules étant destinées à exprimer des propriétés des termes du langage de programmation. Les systèmes de types sont en général constitués de **types**

atomiques (`nat, bool, string` ...), de **règles de formation** des types et de **règles de typage** des expressions.

Les règles de formation vont servir à introduire des connecteurs de types ; la plupart des langages typés disposent des constructeurs produit ou somme de types et du connecteur $\rightarrow$.

Pour former les types du système de types, on considère les termes du premier ordre sur la signature formée des constructeurs et des constantes de types.

Exemple 1 : Le système de types de CAML [18]
CAML dispose de constructeurs de types binaires ($\rightarrow, *$), unaires (`list`), ainsi que de constantes de types (ou types atomiques : `bool, nat`, ...), de variables de types et de deux classes de types :

- les types de base (`int, bool, string,`...),
- les types composés,

La production grammaticale qui les définit est la suivante :

$$\texttt{Type} := \texttt{BasicType} \mid \texttt{Type} \rightarrow \texttt{Type} \mid \texttt{Type} * \texttt{Type}$$

Dans ce langage, les constructeurs de valeurs et les constructeurs de types ne sont pas dénotés de la même manière. Les termes définissant les expressions à évaluer et ceux définissant les types ne sont donc pas construits sur la même signature :

- c'est le constructeur "," qui sert à construire les uplets de valeurs et "$*$" les uplets ou produits de types ;
- les listes de valeurs sont construites par "[ ]" tandis que le type "liste sur un type variable a" s'écrit `'a list` ;
- la valeur "( )" est de type `unit`.

Les règles de typage, souvent présentées sous la forme de règles d'inférence logiques utilisant le symbole $\vdash$, vont comprendre les déclarations de constantes typées (`true` est de type `bool` d'où la règle $\vdash$ `true : bool`) et permettre d'inférer les jugements de la forme

$$C \vdash M : \tau$$

où C est un contexte (ensemble fini de déclarations de variables), M un terme du langage de programmation et τ un type, c'est-à-dire un terme du "langage de typage". Ces règles de typage peuvent être utilisées pour vérifier un type (lorsque C, M et τ sont connus) ou pour une synthèse de type (lorsque τ est inconnu) comme c'est le cas en CAML.

Synthèse de types en CAML : Au problème "Comment trouver le type le plus général pour une expression du langage ?", CAML et les langages de cette famille répondent en utilisant un algorithme basé sur un procédé d'unification "destructive" sur les termes du premier ordre que sont les types.

Un nouvel ensemble de règles d'inférences est écrit (par rapport à celles qui définissent ce qu'est un type), que l'on peut voir comme un algorithme puisqu'il admet exactement une conclusion pour chaque construction syntaxique.

Pour une expression donnée du langage CAML, la déduction de type pour cette expression utilise exactement une règle par sous-expression. Ainsi la déduction est structurée selon le même schéma que celui de l'expression, ce qui permet par là-même de reconstruire une expression à partir de son arbre de déduction de type.

Remarque : Le type `int` $\rightarrow$ `int` est le type exprimant qu'un terme du langage de programmation de ce type est une fonction des entiers naturels vers les entiers naturels.

Le fait de pouvoir écrire des types comme

$$(\texttt{int} \rightarrow \texttt{int}) \longrightarrow (\texttt{int} \rightarrow \texttt{int})$$

nous fait parfois parler de langages d'"ordre supérieur". Cette notion peut se formaliser en se donnant une fonction ord associant à tout type atomique un entier (ex : $ord(\texttt{int}) = 1$), prolongée par

$$ord(\tau_1 \rightarrow \tau_2) = \max(ord(\tau_1) + 1, ord(\tau_2)).$$

Exemple 2 : Les types du système F
Le système F de Girard [14] est un système de types dits de second ordre (cette notion d'ordre n'étant pas à assimiler à la précédente), avec le λ-calcul comme modèle de langage de programmation. Pour former les types,on considère les formules écrites sur un ensemble infini dénombrable de variables (dites variables de type : $X, Y, \ldots$), le connecteur $\rightarrow$ et le quantificateur $\forall$.

On définit sur cet ensemble de formules une notion de substitution, puis de α- équivalence (par induction et en utilisant la substitution : $t_1 \equiv t_2$ si t_2 est obtenu à partir de t_1 en renommant les variables liées de t_1).

Les types du système F sont alors des classes d'équivalence de formules pour la α-équivalence :

$$Id = \forall X(X \rightarrow X) \text{ (type de l'identité)}$$

$$Bool = \forall X\,\{X, X \rightarrow X\} \text{ (type des booléens)}$$

$$Ent = \forall X\,\{(X \rightarrow X) \rightarrow (X \rightarrow X)\} \text{ (type des entiers)}$$

$$A \wedge B = \forall X\,\{(A, B \rightarrow X) \rightarrow X\} \text{ (type produit de A et B)}$$

$$List(Y) = \forall X\,\{(Y, X \rightarrow X), X \rightarrow X\} \text{ (type des listes d'objets de type Y)}$$

Un terme du λ-calcul de type Id sera par exemple $M \equiv \lambda x x$, un λ-terme booléen $M \equiv \lambda x \lambda y x$ ou $M \equiv \lambda x \lambda y y$.

Remarque : Il est possible de "décoder" les types du système F dans la théorie des esquisses, ou plutôt des trames qui sont une généralisation de la notion d'esquisse.

A partir du type $List(Y)$ par exemple, on obtient une trame d'ordre 2 [5, 1].

1.2 Les langages algébriques - LPG

Si les langages fonctionnels déterminent la construction de types à partir d'opérations sur des types déjà définis, par somme ou produit par exemple, les langages algébriques relient la notion de type (de données) à celle de classe de modèles d'une théorie algébrique.

A titre d'exemple, on écrit en LPG [3] des spécifications algébriques et on s'intéresse à la classe des modèles d'une telle spécification. Une **propriété**

LPG est une spécification particulière, définissant une classe de modèles, et on peut déclarer dans LPG qu'un **type**, ou type abstrait algébrique, est un modèle d'une propriété.

Exemple : Soient la propriété

```
property Ensemble
sorts E
end Ensemble
```

et le type abstrait

```
type Naturel
sorts nat
constructors
   zero :  -> nat
   succ :  nat -> nat
operators
   + :  (nat,nat) -> nat
variables i, j :  nat
equations
   1 :  zero + j ==> j
   2 :  succ(i) + j ==> succ(i+j)
end Naturel
```

On spécifie que `Naturel` est modèle de `Ensemble` par la déclaration :

```
models Naturel :  Ensemble[nat]
```

On fait ainsi correspondre aux sortes et opérateurs de la propriété les sortes et opérateur d'un type : ici ce sont seulement les sortes `nat` et `E` qui sont en correspondance.

1.3 Exemple du langage Haskell

Haskell [12] est un langage fonctionnel qui permet de définir des types abstraits algébriques et des classes de types.

Types et valeurs d'expressions ne sont pas confondus mais la notion de **genre** permet de caractériser les types. Les valeurs de types sont construites à partir de constructeurs de types et les expressions de types sont caractérisées par un **genre** pour assurer leur validité. On peut ainsi les classer avec les deux seules formes qui peuvent être prises :

- Le symbole $*$ représente le genre de tous les constructeurs de types 0-aires.
- Si k_1 et k_2 sont des genres, alors $k_1 \to k_2$ est le genre des types qui prennent un type de genre k_1 et retournent un type de genre k_2.

On détermine alors le genre des expressions de types :

- Les variables de types ont leur genre inféré par le contexte dans lequel elles apparaissent.
- Les constructeurs constants `Int, Float, Char, ...` sont de genre $*$.
- Le constructeur $\to$ a pour genre $* \to * \to *$.
- Les constructeurs de uplets de types (), ( ), (,), (, ,), pour 0, 1, 2 ou 3 éléments, ont pour genre $*$, $* \to *$, $* \to * \to *$, $* \to * \to * \to *$.

La construction d'un type abstrait en Haskell peut se faire en utilisant une déclaration `newtype` à partir d'un type existant.

Exemple : Piles définies par les listes

```
module Stack (StkType, push, pop, empty) where
newtype StkType a = Stk[a]
push x (Stk s) = Stk (x :  s)
pop (Stk (x :  s)) = Stk s
empty = Stk [ ]
```

Dans le noyau Prelude de Haskell, on a par exemple les types abstraits `Integer, Float` et la classe prédéfinie de types `Num`, dont `Integer` et `Float` sont des **instances** selon la terminologie Haskell. Ils sont par ailleurs instances de la classe `Eq` définie par :

```
class Eq a where
   (==), (/ =) :: a  → a  → Bool
   x/ = y   =  not (x == y)
```

Une déclaration `instance` stipule qu'un type est une instance d'une classe.

On notera que les types Haskell sont structurés par un ordre de généralisation : on dit qu'un type T_1 est **plus général** qu'un type T_2 (en faisant appel à une notion de substitution) et le type le plus général que l'on peut associer à une expression est appelé son type principal.

1.4 Quelques spécificités d'Aldor

Aldor est un langage basé sur l'évaluation d'**expressions**. On verra ultérieurement quelles sont les règles à respecter pour former les expressions, ie la syntaxe du langage.

Chaque expression du langage est évaluée et cette exécution produit un ensemble de **valeurs**.

Les valeurs calculées peuvent ensuite être sauvées dans des **variables** ou des **constantes** identifiées (un **identificateur** n'étant qu'une chaîne de caractères, exception faite des mots réservés du langage tels `with, add, if, then, else, repeat` ..., utilisé pour nommer des variables ou des constantes par une affectation := ou une définition ==).

Définir une variable ou une constante consiste à associer une expression, un identificateur et éventuellement un type.

Définir une paire $(nom, valeur)$ revient alors à apparier les noms de variables et les valeurs affectées, et un ensemble de paires $(nom, valeur)$ constitue un **environnement**.

L'approche choisie par les concepteurs est celle de "types sur les variables et constantes" : les types sont associés aux variables et constantes soit par des déclarations, soit inférés de l'expression qui les définit, et il n'est pas possible, à partir d'une seule valeur, de retrouver son type.

Les types étant décrits par les mêmes formules que les expressions évaluées par le langage, on parle de types comme valeurs de première classe. On pourra en particulier les passer en arguments de fonctions si leurs types (de ces types) sont adéquats ou encore les affecter à des variables.

Cette spécificité a des conséquences très particulières, que l'on se propose de présenter de suite.

Chapitre 2

Les types d'Aldor

On s'intéresse dans ce chapitre aux pures notions de typage : association d'un type à une expression et sa vérification dans le monde des types. D'autres notions viendront s'y greffer en fin de document, notamment celle de nature.

2.1 Expressions d'Aldor et types

Les expressions d'Aldor sont des termes sur une signature donnée Σ . Notons $EXPR$ l'ensemble des termes du premier ordre sur Σ.

Les termes décrivant les types d'Aldor sont produits à partir d'une signature Σ_T. On note $TYPE$ l'ensemble de ces termes.

L'affirmation

$$\textit{Aldor est un langage typé}$$

signifie que l'on définit ainsi une application

$$type : EXPR \rightarrow TYPE$$

telle que chaque expression a exactement un type.

La seule trace visible de cette application en Aldor réside dans l'affichage en mode interactif : un type est proposé après chaque évaluation. C'est en général le type de l'expression évaluée.

Exemple 1:

```
f :  Integer → Integer == (x:Integer):Integer +-> x + 1
```

est une définition de fonction et *type*(`f`) = `Integer` → `Integer`, où le type est construit à partir du terme type `Integer` et du constructeur de type →.

La première conséquence de la sentence

"Tous les termes Aldor sont des valeurs de premier ordre"

s'applique aux termes décrivant les types :

$$TYPE \subseteq EXPR.$$

Pour la plupart des langages de calcul on ne peut comparer les termes sur Σ et les termes sur Σ_T.

Comme on a vu que la prescription d'un type est permise pour tous les éléments de *EXPR*, la question du typage des types sera alors à étudier.

2.2 Fonctions et objets de base

Si *FONC* désigne l'ensemble des fonctions alors

$$EXPR = FONC \amalg OBJ \amalg TYPE \text{ (réunion disjointe)}$$

avec *OBJ* l'ensemble des objets de base (ni fonctions ni types).

La fonction *type* de *EXPR* vers *TYPE* est donc à définir à la fois sur *FONC*, *OBJ* et *TYPE*.

Un type étant qu'une expression, une fonction peut produire un type : on appelle **constructeur de type** toute fonction Aldor retournant un type (→, `Tuple`, `Cross`, etc pour les prédéfinies, et toutes les fonctions définies par l'utilisateur).

Dans l'exemple 1, le constructeur → est un élément de *FONC* et est aussi un élément de Σ_T puisqu'il retourne le type `Integer` → `Integer` une fois appliqué aux deux types `Integer, Integer`, mais il n'est pas dans *TYPE* puisque $FONC \cap TYPE = \emptyset$.

2.3 Satisfaction de types

La quasi-totalité des langages fonctionnels mettent en oeuvre une vérification de types pour l'application : si f est une fonction de type $A \to B$ où A et B sont des types alors $f(x)$ est une expression valide si et seulement si le type de x est exactement A.

Cette vérification est sensiblement différente en Aldor. On a d'une part : $f(x)$ est une expression valide si le type de x satisfait le type A, et d'autre part cette condition concernant uniquement le typage s'accompagne en fait d'une deuxième condition .

Celle-ci ne concerne pas uniquement le typage, elle fait aussi intervenir le corps de la définition. On y reviendra dans le dernier chapitre de ce travail. On ne s'intéresse pour le moment qu'aux seuls types.

Le langage Aldor définit une relation de satisfaction sur les types et, si l'on sait que toute expression a exactement un type, celui-ci peut en satisfaire plusieurs autres.

Cette relation de satisfaction sera décrite dans ce document par le formalisme des esquisses, au cours du chapitre 4.

Exemple 2 : Le type `(Integer, Integer)` des couples d'entiers satisfait le type `Tuple(Integer)` des uplets d'entier. Cette satisfaction est "automatique" en Aldor (6.4.3).

2.4 Types-domaines et types des domaines

Les types des fonctions et des objets de base sont appelés parfois types-domaines en Aldor, ou le plus souvent domaines. Si *DOM* désigne l'ensemble de tous les domaines alors

$$DOM \subseteq TYPE.$$

Exemple 3 : `Integer` and `Integer` $\to$ `Integer` sont des domaines.

Pour en faire une présentation simple, complétée dans le paragraphe suivant, on peut considérer le domaine des entiers comme la définition des opérateurs accessibles sur les entiers.

Pour répondre maintenant à la question "Quel est le type d'un domaine ?", on pourrait introduire un unique type noté `Type`.

Ce type existe en Aldor et il s'agit d'un domaine, mais ce seul typage par `Type` est insuffisant pour exprimer toute la richesse du système de types d'Aldor. La raison principale est la suivante : puisque $TYPE \subseteq EXPR$, un domaine peut être argument d'une fonction dans un contexte d'application. Si le type de tout domaine est `Type`, on ne peut définir des fonctions que sous la forme :

$$g(T : \texttt{Type}) : \texttt{B} == \ldots$$

Cette fonction g prend un domaine en argument. Il est de fait impossible de placer des restrictions sur les domaines acceptables par la fonction g comme arguments, en termes d'opérateurs disponibles par exemple. Si g est par exemple une fonction de tri d'éléments, qui à partir d'une séquence retourne la séquence triée, le domaine de ces éléments doit fournir un opérateur de comparaison, ce que l'on aimerait signifier dans le typage. Ce concept de type unique pour tous les domaines n'est pas satisfaisant.

En réalité tout domaine a un type qui satisfait `Type` sans être nécessairement `Type`. Ce peut être un type-catégorie.

2.5 Types-catégories et types des catégories

2.5.1 Catégories comme types de domaines

Un type-catégorie ou catégorie est destiné à apporter des restrictions sur les domaines possibles dans un contexte de typage donné.

La conception d'un domaine en Aldor est celle du typage dans les langages algébriques : c'est un type abstrait de données (ADT). Il est constitué d'un type exporté (noté % dans la définition du domaine et qui prend ensuite le nom du domaine une fois exporté) , et d'une collection d'opérations exportées, ce qui détermine une signature hétérogène sous-jacente au domaine.

Exemple 4 : un domaine des λ-termes

```
Lambda-termes :with{
 var :  Variable → %;
 -- Variable est un domaine préalablement défini
```

```
 abs :  (Variable, %) → %;
 apply :  (%, %) → %;}
== add {
var (x:Variable) :  % == x pretend %
abs (x:Variable, M:%) == ...
}
```

Le domaine `Lambda-termes` est un ADT sur la signature multi-sortes

$$\sigma =< \{Variable, \%\}, \{var, abs, apply\} > .$$

La notion de catégorie s'appuie sur celle de signature au sens où l'on reprend les symboles fonctionnels de la signature, munis de leur profil. S'y ajoutent des constructions particulières, en termes d'héritage par exemple, non détaillées ici mais dans le chapitre 7.

Exemple 5 : Une catégorie identifiée (ie à laquelle on a associé un identificateur) de termes

```
Termes == with {
 var :  Variable → %;}
-- Toute variable est un terme
```

La catégorie `Termes` est relative à la signature multi-sortes

$$\sigma_{termes} =< \{Variable, \%\}, \{var\} > .$$

si l'on définit une fonction `modele(D:Termes):Type ==...` alors cette fonction `modele` accepte comme arguments toutes les expressions `D` vérifiant la condition désormais habituelle : le type de `D` satisfait la catégorie `Termes`.

Ainsi `Lambda-termes` est un argument valide pour `modele`, son type étant la catégorie non identifiée (ou anonyme)

```
with{
 var :  Variable → %;
 abs :  (Variable, %) → %;
 apply :  (%, %) → %;}
```

Il y a une inclusion de signatures entre cette catégorie et `Termes`, ce qui constitue un des cas de satisfaction de types, donc cette catégorie anonyme satisfait la catégorie `Termes`.

Pour des domaines très usités, on a par exemple :

- `Integer` de type la catégorie Join(IntegerNumberSystem, Steppable) with { `integer ...`} ;
- `Integer` $\rightarrow$ `Integer` de type la catégorie `with`.

2.5.2 Types des catégories

Si *CAT* est l'ensemble des catégories d'Aldor alors

$$CAT \subseteq TYPE$$

et elles ont elles-mêmes un type. Ce pourrait être là encore un type unique, noté `Category`. Ce type existe en Aldor et c'est un domaine, mais pour de semblables raisons ce n'est pas le type unique des catégories, il n'est qu'un type satisfait par tous les types des catégories.

Dans ce langage, les catégories sont typées par d'autres catégories ou par le domaine `Category`. Une catégorie peut être utilisée comme type d'une autre si une relation d'héritage est satisfaite, par exemple si elle fournit suffisamment d'opérations ou constantes exportées.

Exemple 6 : Les catégories `Group` et `Monoid` sont de type `Category`, et `Group` hérite de `Monoid` puisque :

- `Monoid :  Category == BasicType with` $\{1 : \%; * : \ldots\}$
- `Group :  Category == Monoid with` $\{$`inv` $: \% \rightarrow \%; \ldots\}$

Toutes ces notions seront définies ultérieurement.

2.6 Domaines et catégories comme types

Il est maintenant possible de typer toute expression Aldor par un domaine ou une catégorie.

$$TYPE = CAT \amalg DOM.$$

Les deux domaines `Type` et `Category` ont une fonction privilégiée dans le typage :

$type : FONC \to DOM \backslash \{\texttt{Type,Category}\}$
$type : OBJ \to DOM \backslash \{\texttt{Type,Category}\}$
$type : DOM \to CAT \cup \{\texttt{Type}\}$
$type : CAT \to CAT \cup \{\texttt{Type, Category}\}$

Il n'est alors pas nécessaire d'introduire de nouveaux types puisque les catégories typent les catégories.

Avec ce système de types, une expression E est :

- une fonction ou un objet de base si le type de E est un domaine différent de `Type` et `Category` ;
- une catégorie si le type de E est `Category` ;
- un domaine ou une catégorie sinon, i.e. si le type de E est `Type` ou une catégorie.

Cette dernière ambiguité peut être levée si nécessaire (application de fonction par exemple), en faisant alors référence à la notion de nature et non plus de typage. Ceci sera abordé en 5.1.4 et repris en 7.2.

On notera tout de même l'ambiguité que ce choix introduit en Aldor.

`Type` pouvant être le type d'une catégorie, alors on obtient un cas de satisfaction immédiat :

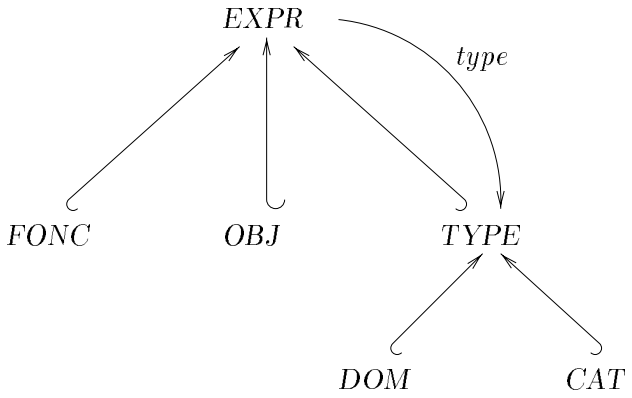
`Type` satisfait `Category`

puisque tout type de catégorie satisfait `Category` (ce sera défini en termes d'esquisses).

Mais `Type` peut aussi être le type d'un domaine. On se trouve donc dans la situation où le type d'un domaine satisfait `Category`, ce qui peut troubler l'utilisateur a priori, mais la deuxième condition de vérification de type que l'on introduira au dernier chapitre assure la validité des expressions que l'on forme.

2.7 Présentation graphique

Les différents ensembles et applications introduits dans ce chapitre se présentent ainsi :

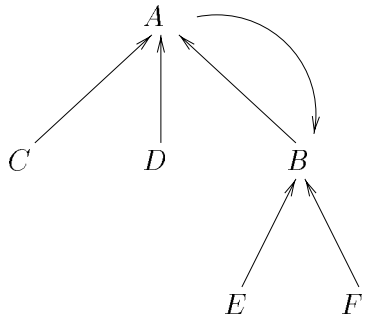


Chapitre 3

Esquisses

Le schéma précédent (fin du chapitre 2) liait des ensembles (*EXPR, TYPE,* ...) par le moyen d'applications dont certaines étaient des injections.

Ce dessin G_2 ne fait que refléter, en termes ensemblistes, une structure que l'on aurait pu décrire de manière plus formelle par le graphe G_1 suivant :



constitué uniquement d'objets et de flèches au sens le plus strict.

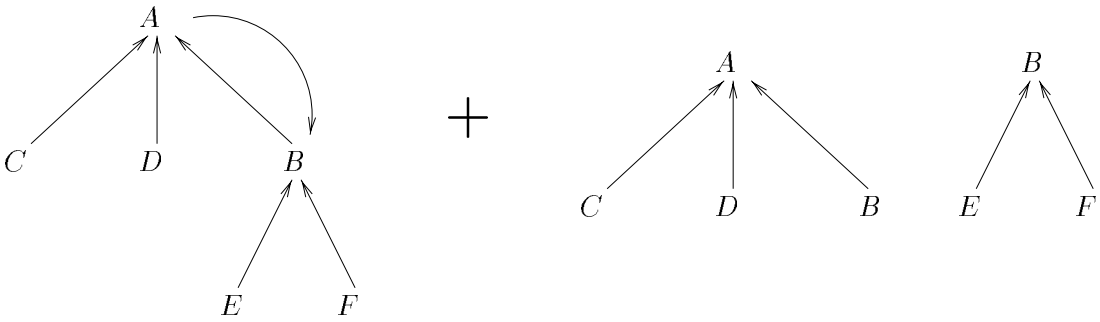
G_1 et G_2 ont la même structure au sens où l'on peut envoyer G_1 sur G_2 par une famille d'applications, objets sur ensembles et flèches sur applications, ici :

- A sur $EXPR$
- B sur $TYPE$
- C sur $FONC$
- etc

en respectant domaine et codomaine de chacune des flèches.

On connaît d'autres propriétés sur ces ensembles qui ne sont pas exprimées dans G_2, par exemple le fait que *EXPR* est la somme des ensembles *FONC, OBJ, TYPE* et *TYPE* la somme de *DOM* et *CAT*.

Par contre ceci peut se faire explicitement à partir de G_1 en décrétant que certains cônes ont un statut particulier, auquel cas on ne considère plus seulement G_1 mais G_1 et un ensemble de cônes distingués issus de G_1 :



Dans le cas présent, les deux cônes distingués sont des cônes inductifs.

Ce graphe G_1^+ avec cônes distingués doit encore pouvoir s'envoyer sur G_2 (on n'a rajouté ni objet ni flèche) avec la même famille d'applications que précédemment. On doit préciser cette fois que l'image de A doit être la réunion disjointe des ensembles images de C, D, B, idem pour l'image de B, ce qui est vérifié par notre famille d'applications, telle qu'elle avait été définie.

La donnée ainsi constituée est celle d'une esquisse (G_1^+), d'un foncteur (la famille d'applications vérifiant une bonne propriété) caractérisant un modèle ensembliste, toutes notions que l'on définit dans ce chapitre.

Remarque : Dans G_1, on peut déjà nommer les objets *EXPR, TYPE, FONC, OBJ, DOM, CAT*, tout ceci étant purement syntaxique et ne présageant en rien des modèles que l'on va considérer.

3.1 Graphes à composition

La notion de graphe est essentielle dans le formalisme que l'on a choisi pour exprimer la théorie des esquisses. Il s'agit en effet d'un formalisme graphique, où le graphe à composition est la base du discours.

Définition 3.1.1 Un **graphe orienté** consiste en la donnée d'une classe d'objets OBJ, d'une classe de flèches FL et des opérations

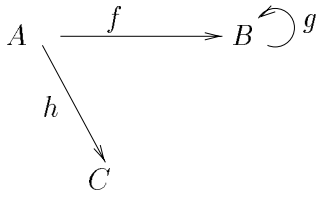
$dom : FL \to OBJ$ (sélection des domaines)

$codom : FL \to OBJ$ (sélection des codomaines)

telles que :

- pour $A \xrightarrow{f} B$ dans FL, $dom(f) = A$ et $codom(f) = B$

Exemple : Sous forme graphique :



ou plus classiquement :
$OBJ = \{A, B, C\}$
$FL = \{A \xrightarrow{f} B, B \xrightarrow{g} B, C \xrightarrow{h} A\}$
$dom(f) = A, dom(g) = codom(g) = B, \ldots$

Définition 3.1.2 Un **graphe à composition (fort)** consiste en la donnée d'un graphe orienté et des opérations :

$selid : OBJ \to FL$ (sélection des identités)

$comp : FL \star FL \to FL$ (composition)

telles que :

- $FL \star FL \subseteq \{(g,f) \in FL \times FL : codom(f) = dom(g)\}$,
  $FL \star FL$ est appelé l'ensemble des couples de flèches composables,

  $$\forall (g,f) \in FL \star FL, comp : (g,f) \mapsto g \circ f$$

  pour reprendre la notation habituelle des composées.

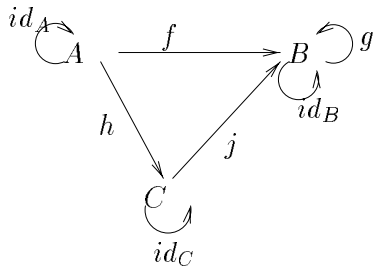
- Axiome de **position** :
  $\forall B \in OBJ,\ dom(selid(B)) = B = codom(selid(B))$, soit encore

  $$dom \ \circ \ selid = codom \ \circ \ selid = Id_{OBJ}$$

  (fonction identité sur la classe OBJ)

- Axiome **d'unitarité** :
  pour tout objet B, la flèche $selid(B)$, notée parfois id_B est une identité :
  $\forall A \xrightarrow{f} B, \forall B \xrightarrow{g} C, id_B \circ f = f$ et $g \circ id_B = g$

Exemple de graphe à composition : Sous forme graphique



ou plus classiquement :
$comp(j,h) = j \circ h = f$

Nota On ne représentera pas toujours les flèches identités.

3.2 Catégories

Définition 3.2.1 Une **catégorie** consiste en la donnée d'une classe d'objets OBJ, d'une classe de flèches FL et des opérations :

$dom : FL \to OBJ$ (sélection des domaines)

$codom : FL \to OBJ$ (sélection des codomaines)

$selid : OBJ \to FL$ (sélection des identités)

$comp : FL \star FL \to FL$ (composition)

telles que :

- pour $A \xrightarrow{f} B$ dans FL, $dom(f) = A$ et $codom(f) = B$
- $FL \star FL = \{(g, f) \in FL \times FL : codom(f) = dom(g)\}$,
  $\forall (g, f) \in FL \star FL, comp : (g, f) \mapsto g \circ f$
- Axiome d'**associativité** :
  $\forall A \xrightarrow{f} B, \forall B \xrightarrow{g} C, \forall C \xrightarrow{h} D, h \circ (g \circ f) = (h \circ g) \circ f$
- Axiome de **position** :
  pour tout objet B, on a une flèche $selid(B)$ notée id_B dans FL
  $dom(id_B) = B$ et $codom(id_B) = B$
- Axiome d'**unitarité** :
  pour tout objet B, la flèche id_B est une identité
  $\forall A \xrightarrow{f} B, \forall B \xrightarrow{g} C, id_B \circ f = f$ et $g \circ id_B = g$

Autrement dit une catégorie est un graphe à composition fort, avec une composition associative et une composabilité maximale.

Exemple : La catégorie **Ens** des ensembles avec comme objets tous les ensembles et comme flèches toutes les applications.

3.3 Foncteurs

Définition 3.3.1 Soient $G = (OBJ_G, FL_G, FL_G \star FL_G, dom_G, \ldots)$ et $G' = (OBJ_{G'}, FL_{G'}, FL_{G'} \star FL_{G'}, dom_{G'}, \ldots)$ deux graphes à composition forts, un **homomorphisme** entre G et G' est un triplet $H = (OBJ(H), FL(H), H \star H)$ où :

$$OBJ(H) : OBJ_G \to OBJ_{G'}$$

$$FL(H) : FL_G \to FL_{G'}$$

$$H \star H : FL_G \star FL_G \to FL_{G'} \star FL_{G'}$$

sont des applications telles que :

- $\forall f \in FL_G$,

$$\begin{cases} dom_{G'}(FL(H)(f)) &= OBJ(H)(dom_G(f)) \\ codom_{G'}(FL(H)(f)) &= OBJ(H)(codom_G(f)) \end{cases}$$

  soit encore l'égalité fonctionnelle

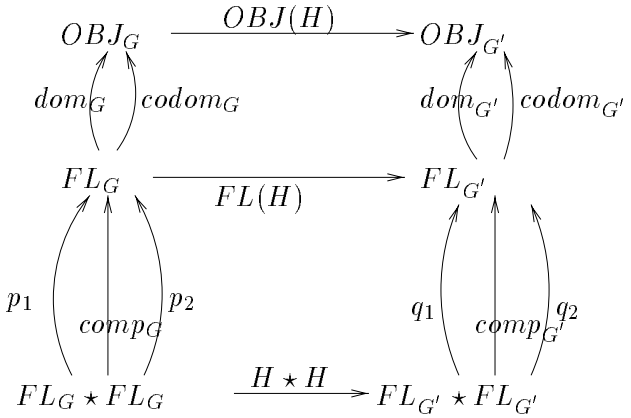
$$\begin{cases} dom_{G'} \circ FL(H) &= OBJ(H) \circ dom_G \\ codom_{G'} \circ FL(H) &= OBJ(H) \circ codom_G \end{cases}$$

  ou la notation simplifiée $\forall f \in FL_G$,

$$\begin{cases} dom(H(f)) &= H(dom(f)) \\ codom(H(f)) &= H(codom(f)) \end{cases}$$

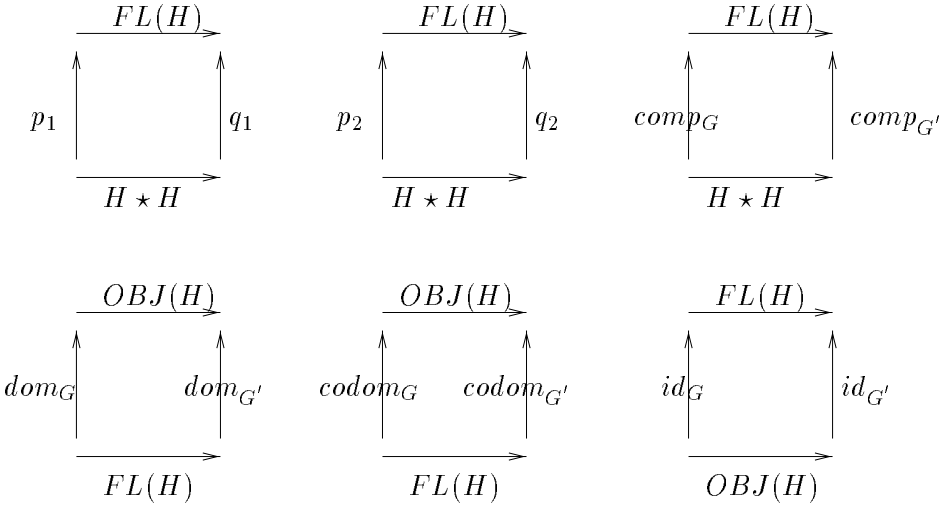
- $H \star H(g, f) = (H(g), H(f))$ en notation simplifiée
- $FL(H)(selid) = selid(OBJ(H))$

Graphiquement :



p_1, p_2, q_1, q_2 sont les projections.

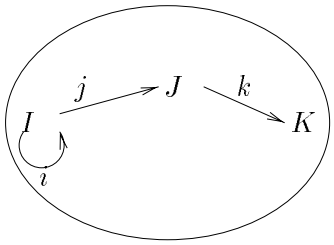
avec les conditions de commutation suivantes sur les diagrammes :



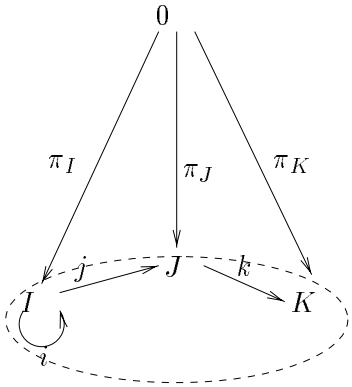
Un homomorphisme entre graphes à composition est appelé un **foncteur**.

3.4 Cônes

Définition 3.4.1 Soit **I** un graphe à composition.



On appelle **cône projectif type** de base **I** le graphe Cp obtenu en ajoutant à **I** un objet (le sommet) et des flèches (les génératrices) en même nombre que les objets de **I**, chacune ayant pour domaine le sommet du cône et pour codomaine un objet de la base **I** :

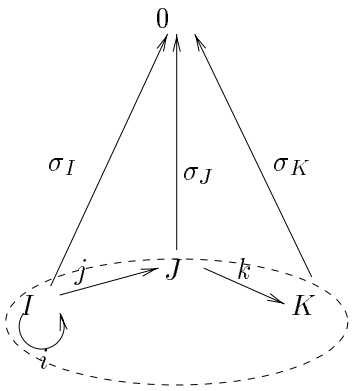


avec une condition de commutation pour chaque flèche de **I**.

$$\begin{cases} \pi_I &= i \circ \pi_I \\ \pi_J &= j \circ \pi_I \\ \pi_K &= k \circ \pi_J \end{cases}$$

Définition 3.4.2 Soit **I** un graphe à composition.

On appelle **cône inductif type** de base **I** le graphe Ci obtenu en ajoutant à **I** un objet (le sommet) et des flèches (les génératrices) en même nombre que les objets de **I**, ayant pour domaine un objet de la base **I** et pour codomaine le sommet du cône :



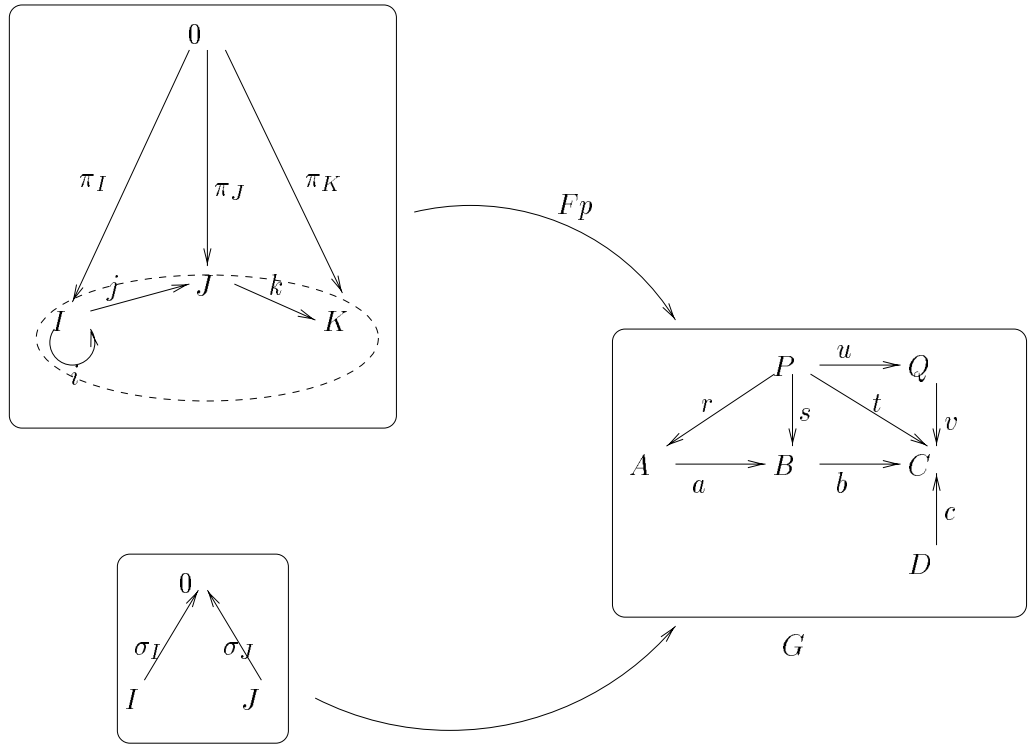
avec là aussi une condition de commutation pour chaque flèche de **I** :

$$\begin{cases} \sigma_I &= \sigma_I \circ i \\ \sigma_I &= \sigma_J \circ j \\ \sigma_J &= \sigma_K \circ k \end{cases}$$

Définition 3.4.3 Soit G un graphe à composition.

Un **cône projectif** (respectivement **inductif**) dans G est un foncteur d'un cône projectif (respectivement inductif) type vers G.

Exemple : Fp **et** Fi



$$Fp : \begin{cases} 0 \mapsto P & j \mapsto a \\ I \mapsto A & k \mapsto b \\ J \mapsto B & \pi_I \mapsto r \\ K \mapsto C & \pi_J \mapsto s \\ & \pi_K \mapsto t \end{cases}$$

$$Fi : \begin{cases} 0 \mapsto C & \sigma_I \mapsto b \\ I \mapsto B & \sigma_J \mapsto v \\ J \mapsto Q & \end{cases}$$

Remarque : En parlant de cône, on confondra souvent le foncteur et son image dans un graphe.

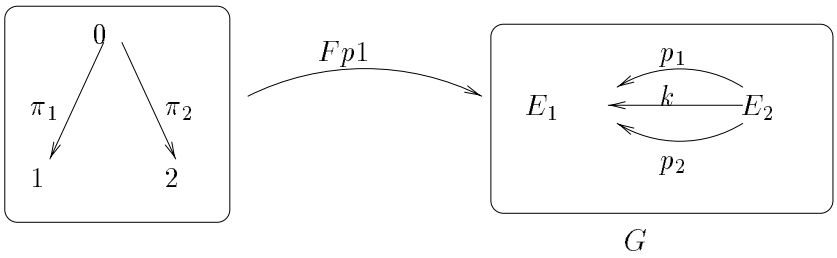
3.5 Esquisses

Définition 3.5.1 Une **esquisse** E consiste en la donnée d'un graphe à composition G (le support de E), d'un ensemble $\mathcal{P}$ de cônes projectifs dans G et

d'un ensemble $\mathcal{I}$ de cônes inductifs dans G.

Ces cônes sont appelés cônes **distingués** dans E.

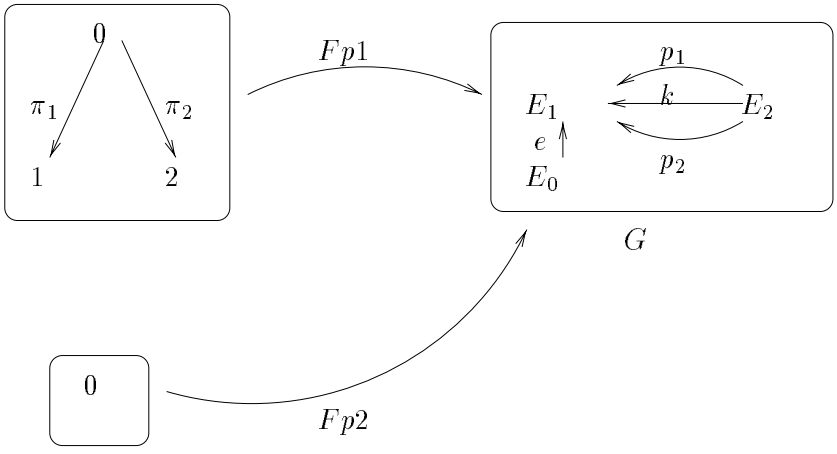
Exemple : Une esquisse E_1 des ensembles munis d'une loi binaire ; $E_1 = (G, \mathcal{P}, \mathcal{I})$ avec :



$$Fp1 : \begin{cases} 0 \mapsto E_2 & \pi_1 \mapsto p_1 \\ 1 \mapsto E_1 & \pi_2 \mapsto p_2 \\ 2 \mapsto E_1 & \end{cases}$$

$$\mathcal{P} = \{Fp1\},\ \mathcal{I} = \emptyset$$

Exemple : Une esquisse E_2 des ensembles munis d'une loi binaire et d'une loi 0-aire (une constante) ; $E_2 = (G, \mathcal{P}, \mathcal{I})$ avec :



$$Fp2 : 0 \mapsto E_0 \text{ (base vide !)}$$

$$\mathcal{P} = \{Fp1, Fp2\},\ \mathcal{I} = \emptyset$$

3.6 Limites projectives et inductives

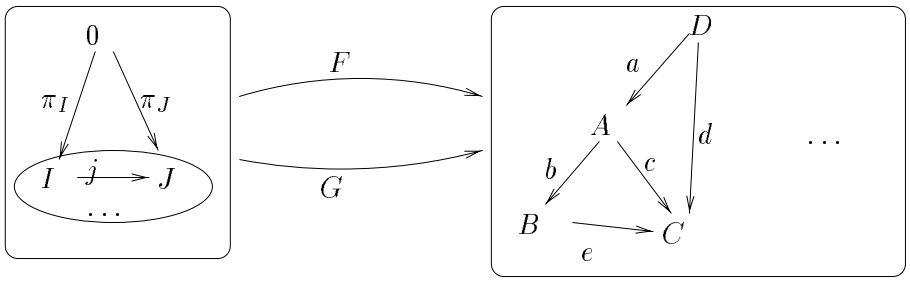
On va donner à certains cônes un statut particulier, celui d'être une limite projective ou inductive.

On n'exprimera pas ces propriétés d'algèbre universelle sur les foncteurs eux-mêmes, ce qui nous amènerait à définir d'autres objets de la théorie des catégories, mais sur leurs images dans un graphe.

Définition 3.6.1 Soient **C** une catégorie et Cp (respectivement Ci) un cône projectif (respectivement inductif) type de base **I**, soit F un foncteur de Cp (respectivement Ci) vers **C**.

$F = (OBJ(F), FL(F), F \star F)$ est un cône limite projective si :

pour tout autre cône G de même base **I** dans le graphe sous-jacent à **C**, il existe une unique flèche dans **C** de $OBJ(G)(0)$ vers $OBJ(F)(0)$ qui réalise les conditions de commutation pour chacune des arêtes des cônes.



C

Supposons :

$$\begin{cases} OBJ(F)(0) = A & FL(F)(j) = e \\ OBJ(F)(I) = B & FL(F)(\pi_I) = b \\ OBJ(F)(J) = C & FL(F)(\pi_J) = c \\ \quad \ldots \quad \ldots & \end{cases}$$

$$\begin{cases} OBJ(G)(0) = D & FL(G)(j) = e \\ OBJ(G)(I) = B & FL(G)(\pi_I) = b \circ a \\ OBJ(G)(J) = C & FL(G)(\pi_J) = d \\ \quad \ldots \quad \ldots & \end{cases}$$

Alors il existe une unique flèche de domaine D et codomaine A, notons la $fact(b \circ a, d)$, qui réalise :

$$\begin{cases} b \circ fact(b \circ a, d) &= b \circ a \\ c \circ fact(b \circ a, d) &= d \end{cases}$$

Certains cônes limite projective sont appelés également cônes produit, ou produits, et parfois identifiés au seul sommet image s'il n'y a pas ambiguité.

Le cas inductif s'obtient en inversant les flèches et un cône limite inductive s'appelle parfois un cône somme ou une somme ; la flèche analogue à $fact$ y est notée $cofact$.

Exemple : Le produit cartésien de deux ensembles muni de ses deux projections est un cône limite projective dans la catégorie Ens des ensembles. La réunion disjointe joue le rôle de la limite inductive.

Cas particulier : Cônes limites projectives de base vide

La propriété générale devient excessivement simple lorsque les cônes considérés sont de base vide. Ainsi un cône limite projective de base vide vers un graphe G sélectionne un objet du graphe, noté $\mathbf{1}$ de coutume, vérifiant :

$$\forall A \in OBJ_G,\ \exists! \lambda_A \in FL_G, dom(\lambda_A) = A, codom(\lambda_A) = 1$$

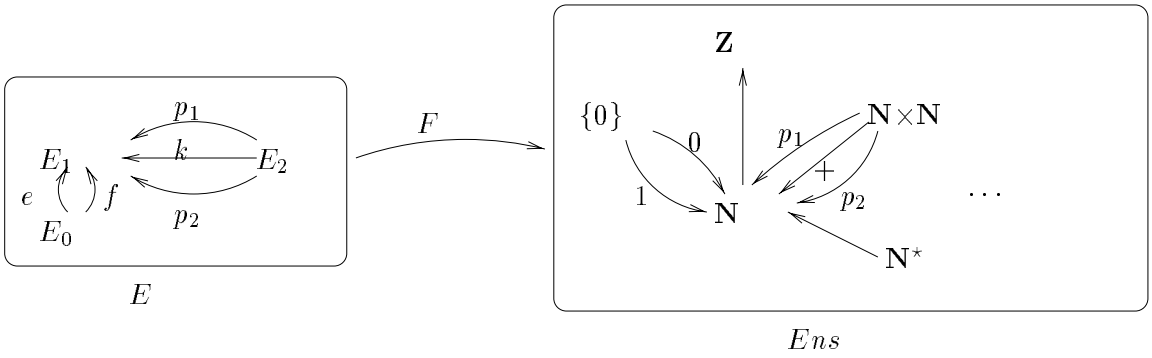
On appelle souvent $\mathbf{1}$ l'objet **terminal** et une flèche de $\mathbf{1}$ vers un autre objet est appelée une **constante**.

3.7 Modèle

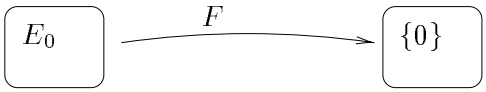
Définition 3.7.1 Soit $E = (G, \mathcal{P}, \mathcal{I})$ une esquisse. Un **modèle** de E dans une catégorie C consiste en la donnée d'un foncteur F de G vers C tel que :

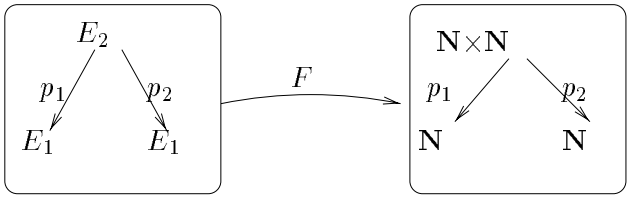
- l'image par F de tout cône projectif distingué dans E est un cône limite projective dans C ;
- l'image par F de tout cône inductif distingué dans E est un cône limite inductive dans C.

Exemple : Un modèle ensembliste d'une esquisse E



où les cônes distingués dans E sont envoyés dans Ens sur des cônes limites comme suit :





Théorème 3.7.1 *Les modèles d'une esquisse E dans une catégorie C forment une catégorie, que l'on notera le plus souvent $Mod(E, C)$, dont les objets sont les modèles de E dans C et les flèches les transformations naturelles entre les modèles.*

3.8 Catégorie cartésienne et cartésienne fermée

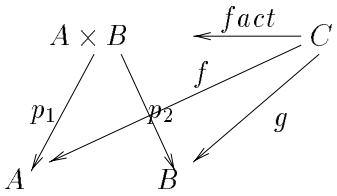
Définition 3.8.1 **C** est une catégorie **cartésienne** si :

1. **C** est une **catégorie**

2. il existe un objet **1**, appelé objet **terminal**, tel que, pour tout objet A, $FL_{A,\mathbf{1}}$ contient un seul élément noté λ_A

3. pour tous objets A et B, il existe un objet $A \times B$ et deux flèches $p_1^{A,B} : A \times B \to A$ et $p_2^{A,B} : A \times B \to B$, tels que, pour tout objet C et toutes flèches $C \stackrel{f}{\to} A$, $C \stackrel{g}{\to} B$, il existe une unique flèche $fact(f,g) : C \to A \times B$ telle que :

$$\begin{cases} p_1^{A,B} \circ fact(f,g) = f \\ p_2^{A,B} \circ fact(f,g) = g \end{cases}$$



En d'autres termes, une catégorie cartésienne est une catégorie qui contient tous les produits finis.

Exemple : La catégorie *Ens* des ensembles munie du produit cartésien.

Exemple : Considérons toutes les formules du premier ordre du calcul des propositions, on obtient une catégorie *Prop* ayant comme objets les propositions et comme flèches les dérivations.

Par exemple, soient P et Q deux propositions, si $P \vdash Q$, alors il existe une flèche dans *Prop* de domaine P et codomaine Q.

En outre, la catégorie *Prop* est une catégorie cartésienne puisque, dès lors que l'on a deux objets P et Q, on a l'objet $P \wedge Q$, et les deux flèches déduites des règles

$$P \wedge Q \vdash P \,, \quad P \wedge Q \vdash Q$$

sont les deux flèches de projection.

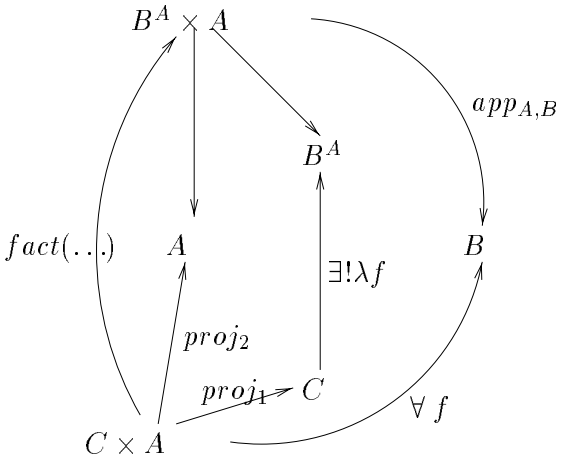
Définition 3.8.2 **C** est une catégorie **cartésienne fermée** si :

1. **C** est une **catégorie cartésienne**

2. pour tous objets A et B, il existe un objet B^A et une flèche $app_{A,B} : B^A \times A \to B$, appelée **application**, tels que,

   pour tout objet C et toute flèche $f : C \times A \to B$, il existe une unique flèche $\lambda f : C \to B^A$ appelée la **curryfiée** de f, telle que :

$$app_{A,B} \circ fact(\lambda f \circ proj_1, proj_2) = f$$



Exemple : La catégorie Ens des ensemble est une catégorie cartésienne fermée. Pour tous objets A et B de Ens, on a l'objet $Hom(A, B)$ ensemble des applications de A vers B qui joue le rôle de l'objet exponentiel B^A et la flèche d'application qui, à une application f de A vers B et un élément a de A associe l'élément $f(a)$ de B.

Exemple : La catégorie $Prop$ est une catégorie cartésienne fermée. Pour tous objets P et Q, on a un objet exponentiel $P \Rightarrow Q$ et une flèche d'application associée déduite de la règle

$$(P \Rightarrow Q) \wedge P \vdash Q.$$

3.9 Catégorie-type

Définition 3.9.1 Soit $U : \mathbf{C} \to \mathbf{C}'$ un foncteur entre deux catégories.

Soient C un objet de $\mathbf{C}$, C' un objet de $\mathbf{C}'$ et ε une flèche de C' vers $U(C)$.

On dit que ε présente C comme un **objet libre engendré** par C' relativement à U si, pour tout objet D de $\mathbf{C}$, pour toute flèche f de C' vers $U(D)$, il existe une et une seule flèche $h : C \to D$ dans $\mathbf{C}$ telle que

$$U(h) \circ \varepsilon = f.$$

(on a noté $U(h)$ pour $FL(U)(h)$.)

$$C \xrightarrow{\exists! h} D$$

$$\begin{array}{ccc} U(C) & \xrightarrow{U(h)} & U(D) \\ \varepsilon \uparrow & \nearrow_{\forall f} & \\ C' & & \end{array}$$

Exemple : Soient Σ une signature unisorte et X un ensemble, l'algèbre $T_\Sigma[X]$ des termes du premier ordre sur Σ et à variables dans X est engendrée librement par X. Ici c'est la catégorie des ensembles qui joue le rôle de $\mathbf{C}'$, et la catégorie des Σ-algèbres le rôle de $\mathbf{C}$.

On peut également appliquer la définition précédente avec $\mathbf{C}$ la catégorie des catégories cartésiennes fermées à limites projectives et inductives, et $\mathbf{C}'$ la catégorie des esquisses.

Une esquisse engendre librement une catégorie cartésienne fermée à limites et U est un foncteur d'oubli qui permet de "voir" une catégorie comme une esquisse.

C'est un théorème dû à C. Ehresmann [8] :

Théorème 3.9.1 *Toute esquisse engendre librement une catégorie cartésienne fermée à limites projectives et inductives, unique à isomorphisme près.*

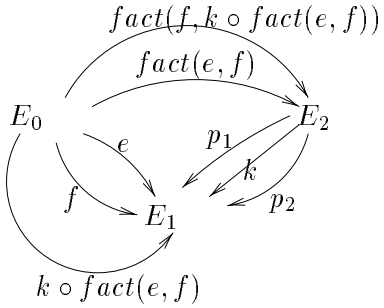
Définition 3.9.2 La catégorie cartésienne fermée à limites engendrée librement par une esquisse $E = (G, \mathcal{P}, \mathcal{I})$ est appelée **catégorie-type** de E.

Pratiquement, on considère le graphe G sous-jacent à E, auquel on ajoute toutes les composées de flèches consécutives dans G, les limites projectives et inductives, les objets exponentiels, les flèches d'application et de curryfication ainsi que toutes les flèches de factorisation (cas inductif et projectif) des cônes distingués dans E.

Les cônes distingués dans E deviennent ainsi des cônes limites dans la catégorie-type de E.

Exemple : Une approche ultra-simplifiée des entiers naturels

La catégorie type de l'esquisse E du 3.7 permet, en rajoutant seulement les factorisations et de la composition pour ce qui nous intéresse, d'atteindre les termes $k(e,f)$, $k(f,k(e,f))$, ...



Ainsi un entier naturel peut être vu classiquement comme un élément de l'ensemble $\mathbf{N}$ dans un modèle de l'esquisse E, ou plus syntaxiquement comme une flèche de E_0 vers E_1 dans l'esquisse E ou dans sa catégorie type.

Ce point de vue "syntaxique" est celui de toute la théorie des esquisses.

Chapitre 4

Une interprétation d'Aldor en termes d'esquisses

Aldor étant un langage typé de calcul formel, basé sur l'interprétation d'expressions, on se retrouve dans un univers comprenant des classes : classe des types, classe des fonctions, ..., et plus généralement classe des expressions, entre lesquelles on met en évidence des liens, par les notions d'héritage ou de typage par exemple.

Une des questions qui se posent est alors de savoir s'il est possible de structurer l'"univers" que compose le langage Aldor, en utilisant la structure essentielle de catégorie entre autres.

Pour cela, on va déterminer une esquisse associée au langage Aldor. Ce sera pour nous une esquisse "généralisée", au sens où on permet la construction d'objets exponentiels dans l'esquisse. Elle est destinée à exprimer aussi bien la syntaxe du langage que les propriétés qui lui sont associées (sous-typage, héritage, ...).

On notera cette esquisse $\mathbf{E}_{Aldor}$, et la catégorie cartésienne fermée à limites qu'elle engendre librement (cf 3.9) sera notée $\mathbf{C}_{Aldor}$. On décrit $\mathbf{E}_{Aldor}$ par étapes, en traduisant systématiquement le fonctionnement d'Aldor en termes d'esquisses.

L'objectif ultime est de retrouver le langage Aldor comme modèle de cette esquisse, en considérant en particulier des modèles ensemblistes pour lesquels les objets de $\mathbf{E}_{Aldor}$ sont interprétés respectivement en la classe des fonctions d'Aldor, la classe des types d'Aldor, etc.

4.1 Structure de catégorie

Proposition 4.1.1 $\mathbf{E}_{Aldor}$ *contient une esquisse des catégories.*

Pour esquisser Aldor en conservant notre point de vue qui est de décrire au moins le typage du langage, on va poser les premiers objets de cette esquisse ainsi que les premières flèches, en tout premier lieu l'objet des types d'Aldor, que l'on note *TYPE*.

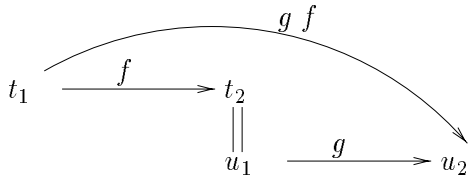
Une possibilité qu'offre Aldor pour agir sur les types consiste à utiliser les fonctions.

Celles-ci permettent en effet, à partir d'un objet d'un certain type, de déterminer un objet d'un type éventuellement différent par application de ladite fonction. On peut donc les caractériser par un type "source" et un type "but", ce qui se représente, en notant *FONC* l'objet des fonctions :

$$TYPE \underset{but}{\overset{source}{\rightleftarrows}} FONC$$

Définition 4.1.1 *Deux fonctions en Aldor sont composables dès lors que la source de l'une et le but de l'autre coïncident :*

si $source(f) = t_1$, $but(f) = t_2$, *si* $source(g) = u_1$, $but(g) = u_2$ *et si* $u_1 = t_2$, *alors la composée* $compose(g, f)$, *notée* $g\ f$ *est une fonction de source* t_1 *et de but* u_2, *ce que l'on peut représenter graphiquement :*

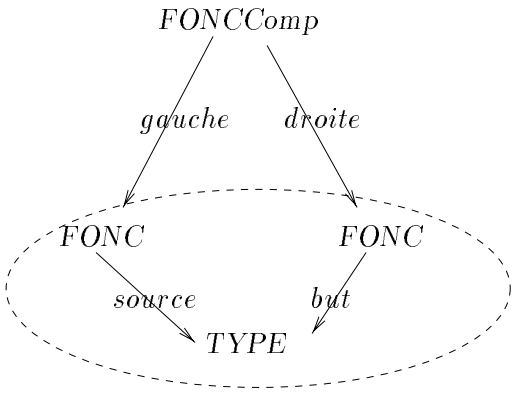


Si l'on considère les couples de fonctions composables, dont l'objet est noté *FONCComp* dans $\mathbf{E}_{Aldor}$, on obtient le graphe à composition suivant sous-jacent à $\mathbf{E}_{Aldor}$:

$$TYPE \underset{but}{\overset{source}{\rightleftarrows}} FONC \underset{droite}{\overset{gauche}{\overset{compose}{\rightleftarrows}}} FONCComp$$

$$but \circ droite = source \circ gauche$$

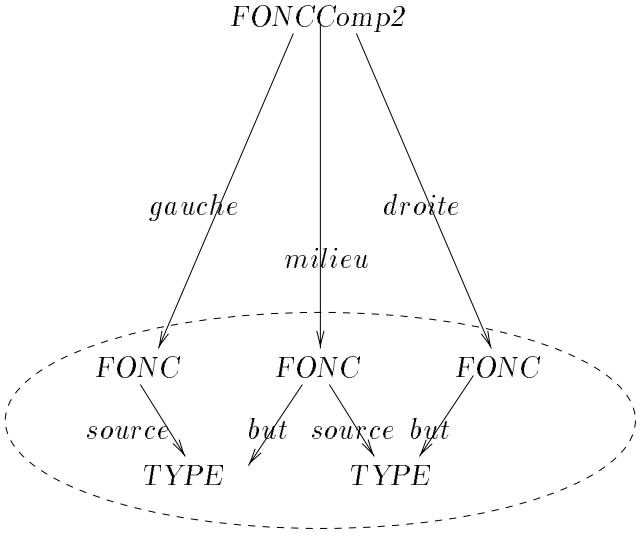
où le cône projectif ci-dessous est distingué dans $\mathbf{E}_{Aldor}$:



La fonction obtenue par composabilité doit respecter les équations suivantes, inscrites dans $\mathbf{E}_{Aldor}$:

$$\begin{cases} source \circ compose & = & source \circ droite \\ but \circ compose & = & but \circ gauche \end{cases}$$

La composition est associative, ce que l'on traduit en introduisant l'objet *FONCComp2* des triples de fonctions composables, sommet d'un cône projectif distingué de $\mathbf{E}_{Aldor}$:



$$\begin{cases} source \circ gauche &= but \circ milieu \\ source \circ milieu &= but \circ droite \end{cases}$$

plus un certain nombre de flèches et d'équations pour exprimer les "projections" partielles et l'égalité des compositions selon le parenthésage.

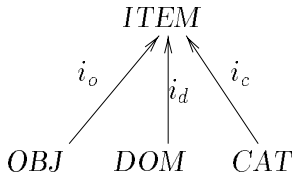
Toutes ces propriétés font de $\mathbf{E}_{Aldor}$ en particulier une esquisse des catégories. [1]

4.2 Fonctions, items et types

4.2.1 Fonctions et items

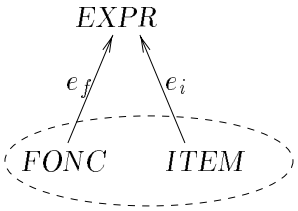
Outre les fonctions, Aldor manipule des objets de base, domaines et catégories, toutes données que l'on désigne sous le vocable "items".

Donc, si *OBJ* désigne l'objet des objets de base d'Aldor, *DOM* l'objet des domaines et *CAT* l'objet des catégories d'Aldor,



et ce cône inductif est distingué dans $\mathbf{E}_{Aldor}$ (donc cône limite inductive dans $\mathbf{C}_{Aldor}$).

On introduit maintenant l'objet des expressions d'Aldor :



[1] On ne fait pas figurer dans le graphe sous-jacent à $\mathbf{E}_{Aldor}$ les flèches identités ni la flèche *selid* de l'esquisse des catégories.

ce dernier cône faisant partie des cônes projectifs distingués dans $\mathbf{E}_{Aldor}$.

Il existe une systématique de traduction en Aldor , non automatique cependant, d'un item vers une fonction :

l'item est $x : T == E$ et la fonction $\tilde{x} : () \to T == () : T$`+->`$E$, la propriété de comparaison étant $\tilde{x}() = x$.

$$FONC \xleftarrow{\ i_f\ } ITEM$$

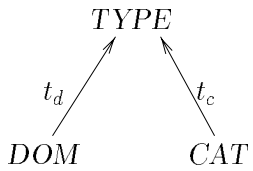
$$source \circ i_f = (\,) \circ \lambda_{ITEM}$$

où ( ) est le "type vide". On utilisera ici ce type comme une donnée, il est en fait produit par un constructeur de types, ce qui sera justifié dans la section à suivre.

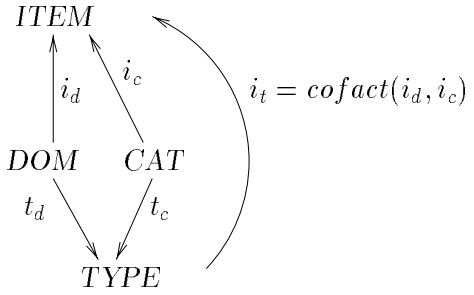
La flèche λ_{ITEM} est l'unique flèche de $ITEM$ vers $\mathbf{1}$ (objet terminal dans la catégorie $\mathbf{C}_{Aldor}$).

4.2.2 Types et items

Les types d'Aldor ne pouvant être que des domaines ou des catégories, on distingue dans $\mathbf{E}_{Aldor}$ un nouveau cône inductif :



Cette construction est dans $\mathbf{C}_{Aldor}$ un cône limite inductive et la propriété universelle de cette limite a pour conséquence :



$$\begin{cases} i_t \circ t_c = i_c \\ i_t \circ t_d = i_d \end{cases}$$

Exemple : Etant donné `Integer` un domaine des entiers en Aldor, on peut le considérer dans $\mathbf{C}_{Aldor}$ comme :

- effectivement un domaine, soit une flèche de $\mathbf{1}$ vers DOM :

$$\mathbf{1} \xrightarrow{\texttt{Integer}} DOM$$

  C'est ce qu'il est naturellement.

- un type, soit une flèche de $\mathbf{1}$ vers $TYPE$, en le composant :

$$\mathbf{1} \xrightarrow{\texttt{Integer}} DOM \xrightarrow{t_d} TYPE$$

- un item, soit une flèche de $\mathbf{1}$ vers $ITEM$:

$$\mathbf{1} \xrightarrow{\texttt{Integer}} DOM \xrightarrow{i_d} ITEM \qquad DOM \xrightarrow{i_t \circ t_d} ITEM$$

selon l'utilisation.

En particulier, si l'on désire typer un objet de base d'Aldor comme un entier, on se servira de la flèche de codomaine $TYPE$, par exemple pour réaliser la déclaration :

```
x :  Integer == 2
```

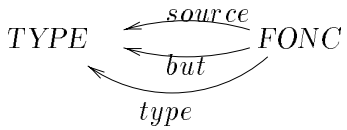
Pour passer ce domaine en argument d'une fonction, on pourra voir `Integer` comme une flèche de codomaine $ITEM$ selon la déclaration de typage qui aura été inscrite dans l'écriture de la fonction appelante :

```
f :  Type→Type == (x:Type):Type +-> E
```

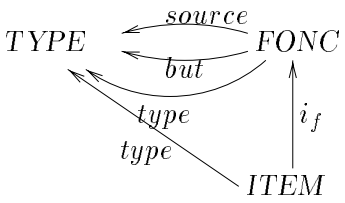
On peut alors envisager d'appliquer la fonction `f` à l'item `Integer` : `f(Integer)`....

4.2.3 Typage

Le point de vue que l'on a adopté pour traduire Aldor, qui est un langage typé, impose de faire figurer dans $\mathbf{E}_{Aldor}$ la notion de typage des fonctions par une flèche *type* de *FONC* verc *TYPE* :



ainsi que celle de typage des items :



avec l'équation additionnelle : $but \circ i_f = type$. [2]

Exemples :

- $type$(`x`) = `Integer`
- $type$(`f`) = `Type` $\rightarrow$ `Type`

Ce typage des items a pour utilité particulière le typage des types : c'est la flèche $type \circ i_t$, de source *TYPE* et de but *TYPE*, qui détermine le type d'un type.

4.2.4 Satisfaction de types

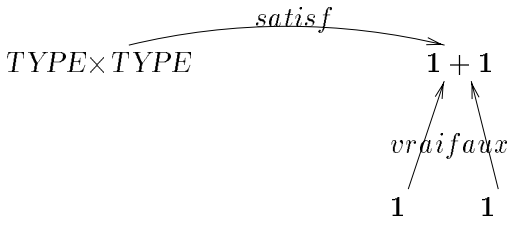
On sait que toute fonction et tout item ont un type unique. A ce type unique, on associe une famille de types constituée de tous les types qui le satisfont. Cette notion de satisfaction de types est à intégrer dans $\mathbf{E}_{Aldor}$.

[2] On notera que $type \circ i_f \neq type$.

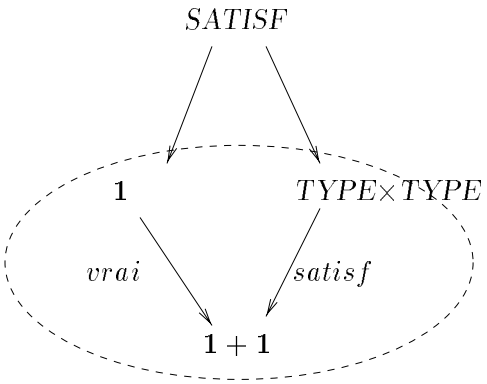
La relation de satisfaction sur les types se traduit ici comme un prédicat binaire, avec la propriété de transitivité (cf chapitre 7). On introduit l'objet *SATISF*, destiné à représenter les couples de types dont le premier satisfait le second.

Exemple : Le type `Category` satisfait le type `Type`.

Le prédicat binaire de la satisfaction de types se représente dans cette approche par une flèche *satisf* de source $TYPE \times TYPE$ et de but $\mathbf{1}+\mathbf{1}$ (objet sommet d'un cône limite inductive dans $\mathbf{C}_{Aldor}$) :



ce qui permet de spécifier l'objet *SATISF* comme sommet d'un cône projectif distingué de $\mathbf{E}_{Aldor}$:

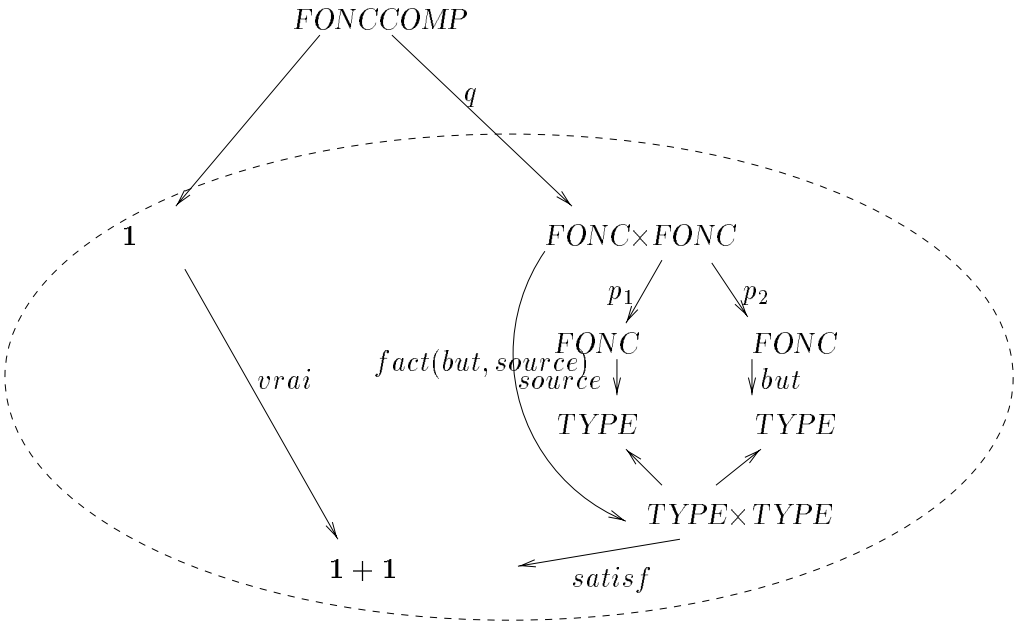


Une autre présentation en sera faite en 5.6, avec quelques exemples moins triviaux.

4.3 Structure de catégorie à composition faible

La notion de composition abordée en début de chapitre (cf déf 4.1.1) n'est qu'une restriction de la composition en Aldor. Les fonctions sont en effet composables dès lors que le but de l'une satisfait la source de l'autre. Le cas où ces deux types sont identiques n'est donc qu'une situation bien particulière pour la composition en Aldor.

L'objet des couples composables faiblement $FONCCOMP$ de $\mathbf{E}_{Aldor}$ est donc sommet d'un cône projectif distingué, mais la base de ce cône n'est plus seulement celle qui avait été décrite précédemment pour $FONCComp$:



Cette transformation s'applique de même à l'objet $FONCCOMP2$ pour lequel on "recopie" en deux exemplaires le cône projectif distingué précédent.

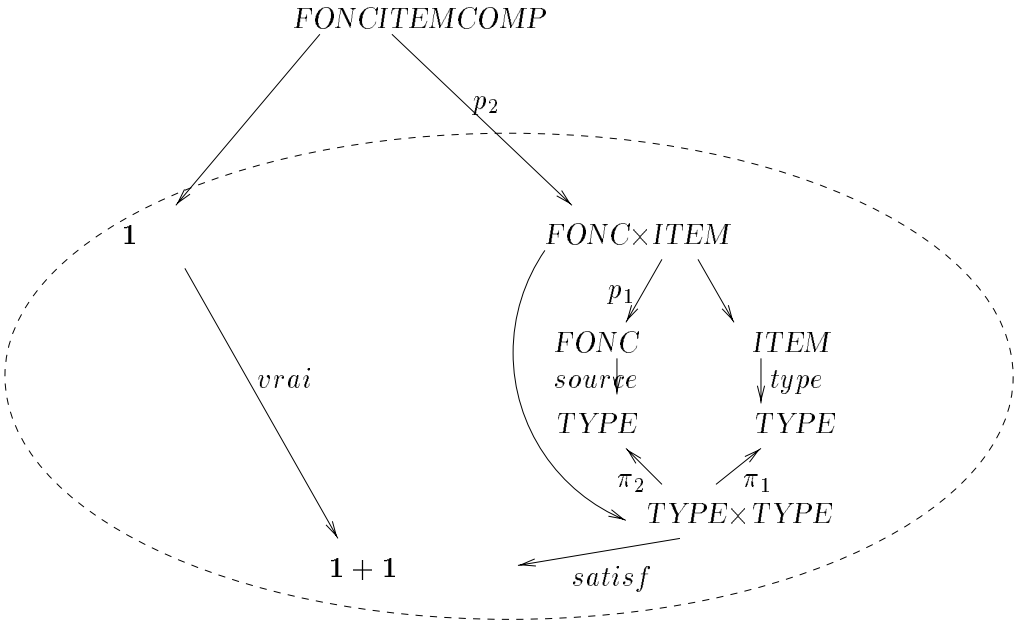
On conserve une flèche de composition $comp$ de $FONCCOMP$ vers $FONC$ associée à quelques équations assurant la cohérence de cette composition : étant données deux fonctions f et g composables faiblement, de composée $g \circ f$, on doit s'assurer que la source de $g \circ f$ est la source de f et le but de $g \circ f$ est le but de g.

Ceci peut s'exprimer plus simplement sans variables :

$$\begin{cases} source \circ comp &= source \circ p_2 \circ q \\ but \circ comp &= but \circ p_1 \circ q \end{cases}$$

En parallèle à la composition de fonctions, on se doit de traduire également l'application d'une fonction à un item .

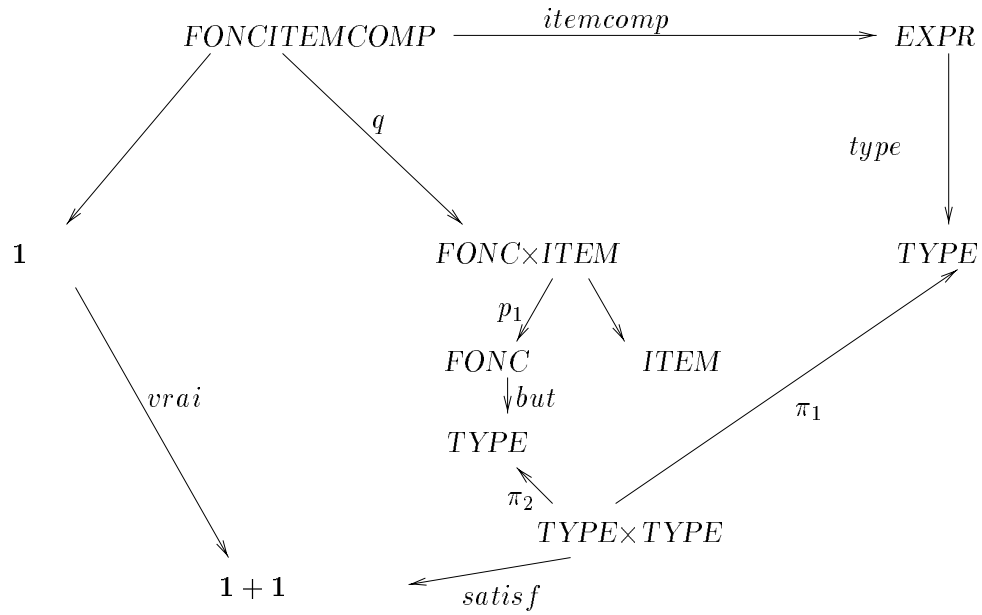
On introduit pour cela le nouvel objet de $\mathbf{E}_{Aldor}$ *FONCITEMCOMP* comme sommet d'un cône projectif distingué dans $\mathbf{E}_{Aldor}$:



FONCITEMCOMP est en outre sommet d'un cône projectif distingué de même indexation que celui définissant *FONCCOMP*, on dispose donc d'une flèche de factorisation que l'on notera i_f^c, de *FONCITEMCOMP* vers *FONCCOMP*.

L'application d'une fonction à un item est ainsi traduite par la flèche $comp \circ i_f^c$, qui retourne donc une fonction de source ().

Plus précisément, on utilise une flèche *itemcomp* pour traduire le résultat de l'application d'une fonction à un item, la nature du résultat étant déterminée par l'analyse du corps de la fonction (cf. 7.2), et son type devant évidemment satisfaire le type-sortie de la fonction utilisée :



$$satisf \circ fact(type \circ itemcomp, but \circ p_1 \circ q) = vrai \circ \lambda_{FONCITEMCOMP}$$

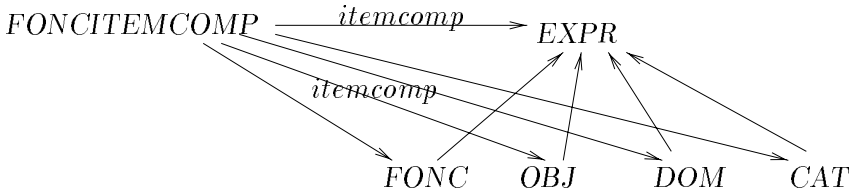
Appliquer une fonction à un item est équivalent à composer ladite fonction avec ledit item transformé en fonction, puis à appliquer la fonction obtenue à l'item ( ) :

$$\begin{aligned} & itemcomp \\ = \;& itemcomp \circ fact(id_{\mathbf{1}} \circ \lambda_{FONCITEMCOMP}, fact(comp \circ i_f^c, i_t \circ () \circ \lambda_{FONCITEMCOMP})) \end{aligned}$$

Si $f : A \to B$ est la fonction et $x : C$ l'item, avec C satisfaisant A, l'item transformé en fonction est $y : () \to C$, avec $y() = x$, et on exprime par cette équation que :

$$(fy)() = f(x)$$

Remarque : *EXPR* étant sommet d'un cône inductif distingué de $\mathbf{E}_{Aldor}$, on va utiliser quatre flèches toutes dénotées *itemcomp*, et la notion de nature en fin de document justifiera ce choix.



L'esquisse $\mathbf{E}_{Aldor}$ ainsi obtenue, et plus particulièrement la sous-esquisse traduisant l'aspect fonctionnel du langage, est une esquisse des catégories à composition faible. La partie de $\mathbf{E}_{Aldor}$ s'articulant autour exprime l'aspect typé de ce langage.

Une spécification plus détaillée de ce que sont les flèches de typage et la manière dont on peut les définir en fonction de flèches plus "élémentaires" seront envisageables à l'issue de ce chapitre, notamment en ce qui concerne le typage des catégories, des objets de base et des fonctions (voir chapitre 6).

4.4 Structure de catégorie cartésienne

Cette section trouve son utilité dans le traitement des conversions automatiques (6.4.3) avec les fonctions prédéfinies `Tuple`, `Cross`, $(\ldots,\ldots)$, relatives aux uplets et aux produits de types.

Tous les ingrédients présentés ici seraient nécessaires pour qui voudrait expliquer complètement dans $\mathbf{E}_{Aldor}$ les régles de conversion automatique.

Proposition 4.4.1 $\mathbf{E}_{Aldor}$ *contient une esquisse des catégories cartésiennes.*

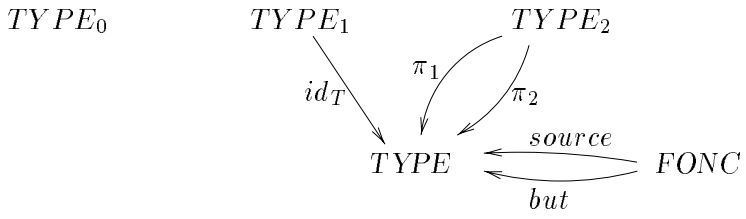
4.4.1 Uplets de types

L'enrichissement de la structure de catégorie à composition faible par l'ajout des produits nécessite au préalable de se donner les n-uplets de types ($n \in \mathbf{N}$). Les objets correspondants dans $\mathbf{E}_{Aldor}$ sont notés $TYPE_n$.

Les objets $TYPE_n$ sont des sommets de cônes projectifs distingués ayant pour arêtes n flèches $\pi_i, 1 \leq i \leq n$, de projection de $TYPE_n$ vers $TYPE$. [3]

[3] On devrait en toute rigueur y faire également figurer n flèches de $TYPE_n$ vers $FONC$, chacune d'entre elles sélectionnant une identité, mais ces flèches ne sont pas réellement nécessaires à notre propos.

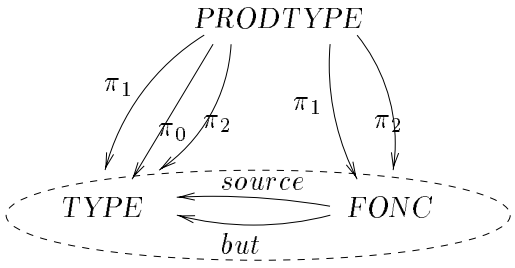
Dans $\mathbf{C}_{Aldor}$, on a de même les sommets $TYPE \times \cdots \times TYPE$ des cônes produits de n exemplaires de $TYPE$, isomorphes aux objets $TYPE_n$ par unicité de la limite projective. Plus particulièrement, $TYPE_1 = TYPE$ et l'on notera ici $TYPE_0$ pour $\mathbf{1}$, parmi ceux que l'on utilisera le plus fréquemment ;on note id_T l'unique projection de $TYPE_1$ vers $TYPE$.



4.4.2 Produits de deux types

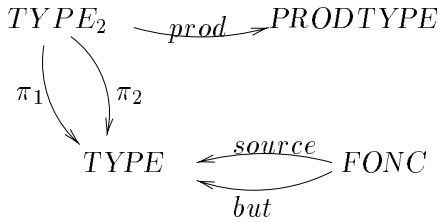
On introduit l'objet $PRODTYPE$ dans $\mathbf{E}_{Aldor}$ pour représenter les produits de deux types, ou plus exactement les cônes produits de deux types.

$PRODTYPE$ est sommet d'un cône projectif distingué dans $\mathbf{E}_{Aldor}$:

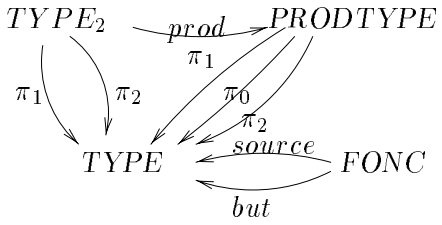


$$\begin{cases} source \circ \pi_1 = \pi_0 \\ source \circ \pi_2 = \pi_0 \end{cases}$$

Le constructeur de produits est symbolisé par la flèche $prod$ de $TYPE_2$ vers $PRODTYPE$:

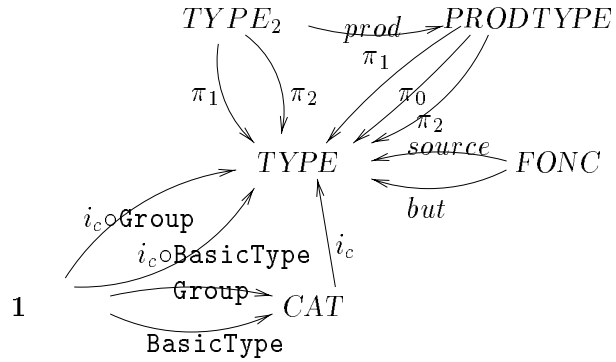


D'un produit de deux types, on dispose des composantes de ce produit ainsi que de ce produit considéré comme un type simple, grâce aux trois projections :



$$\begin{cases} \pi_1 \circ prod = \pi_1 \\ \pi_2 \circ prod = \pi_2 \end{cases}$$

Exemple Dans $\mathbf{C}_{Aldor}$, on a la situation suivante :



$\pi_0 \circ prod \circ fact(i_c \circ Group, i_c \circ BasicType)$ est une flèche de $\mathbf{1}$ vers $TYPE$, c'est cette flèche qui "représente" le type noté `(Group, BasicType)` en Aldor, produit des deux types `Group` et `BasicType`

Il est donc utilisable dans tout contexte de déclaration de type, par exemple :

```
x : (Group, BasicType) :=...
```
[4]

[4]La définition de constantes n'obéit pas aux mêmes règles et leur utilisation est beaucoup plus problématique.

Il est toujours possible, à l'aide de "keyword arguments", de récupérer en Aldor les deux composantes d'un produit de types :

```
(x : Group, y : BasicType) := (Float, Integer)
```

L'appel de `x` (respectivement `y`) permet d'obtenir le type `Float` (respectivement `Integer`), tout comme la flèche π_1 (respectivement π_2) de $\mathbf{E}_{Aldor}$.

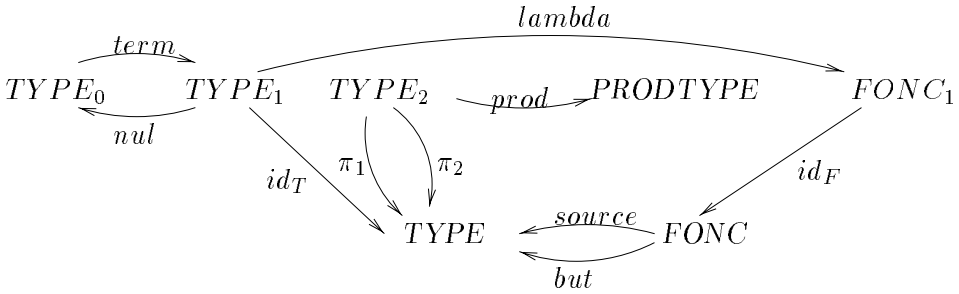
Au produit est associée une factorisation (cf. chapitre 3) exprimée par une flèche *factorise* de *PRODTYPE* vers *FONC*. On ne détaille pas ici la propriété de factorisation ni les équations qui la traduisent dans $\mathbf{E}_{Aldor}$ dans le cas du produit de deux types.

4.4.3 Produit de "rien"

Le cas du produit de "rien" nécessite un traitement particulier. C'est en effet un objet et non un cône non trivial, et cet objet est terminal dans toute catégorie cartésienne.

On fait donc figurer dans $\mathbf{E}_{Aldor}$ une flèche *term* de construction du produit de rien par sélection d'un type, ainsi qu'une flèche *lambda* qui exprime que de tout type A part une flèche ou plutôt une fonction λ_A vers le type vide qui est terminal.

On a de même une flèche *nul* qui envoie un type sur une configuration "type vide".



On doit spécifier l'action de la flèche *lambda* par des équations supplémentaires :

$$\begin{cases} source \circ id_F \circ lambda = id_T \\ but \circ id_F \circ lambda = id_T \circ term \circ nul \end{cases}$$

En toute rigueur, on devrait spécifier également que la fonction sélectionnée par *lambda* est unique, ce qui nécessite quelques flèches et équations de plus (cf).

On est maintenant en mesure de justifier l'utilisation que l'on faisait du type "vide" dans la définition des items au début de ce chapitre. La flèche $(\,) : \mathbf{1} \to TYPE$ utilisée, qui est en fait la composée $:\mathbf{1} \stackrel{()}{\to} DOM \stackrel{t_d}{\to} TYPE$, où le domaine () est une donnée de base d'Aldor, est en fait égale à $id_T \circ term$, autrement dit la sélection du type produit de rien.

4.5 Structure de catégorie cartésienne fermée

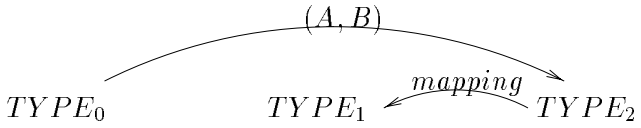
Cette section sera moins détaillée que la précédente puisqu'il n'existe pas en Aldor de formes prédéfinies relatives à la curryfication de fonctions, mais uniquement une forme syntaxique appropriée pour faciliter l'usage des fonctions curryfiées.

Proposition 4.5.1 $\mathbf{E}_{Aldor}$ *contient une esquisse des catégories cartésiennes fermées.*

4.5.1 Exponentielle

Disposant de deux types A et B en Aldor, on peut former le type $A \to B$ par application de la fonction "$\to$", laquelle retourne un domaine Aldor qui est un type de fonctions.

On exprime cette propriété dans $\mathbf{E}_{Aldor}$ par l'ajout d'un constructeur *mapping* :



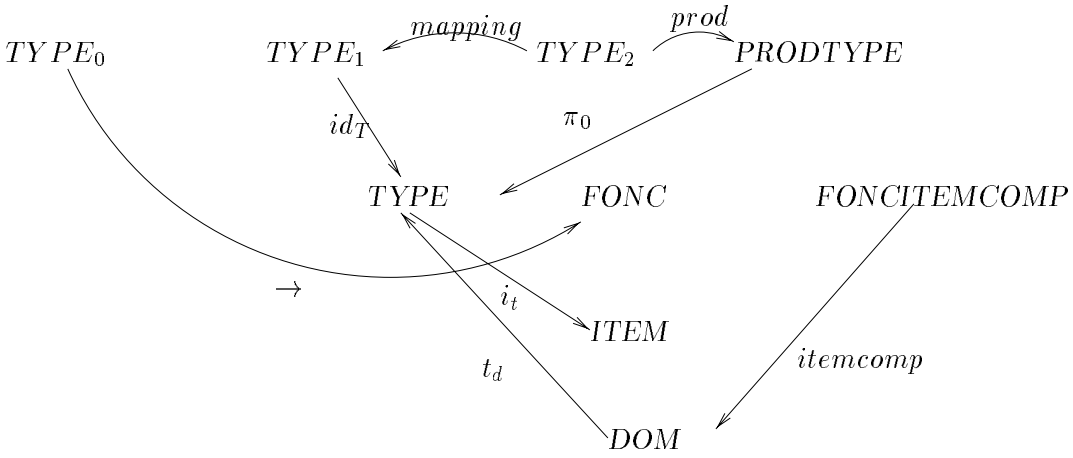
où la flèche $mapping \circ (A, B)$ désigne l'objet "exponentielle de A et de B" au sens des catégories cartésiennes fermées.

On identifie ensuite cet objet exponentiel au type produit par "$\to$" :

$id_T \circ mapping$

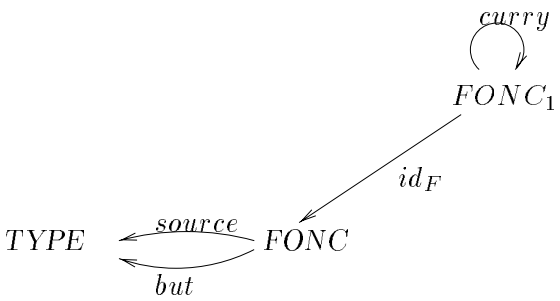
$= \quad t_d \circ itemcomp \circ fact(id_{\mathbf{1}} \circ \lambda_{ITEM}, fact(\rightarrow \circ \lambda_{ITEM}, id_{ITEM})) \circ i_t \circ \pi_0 \circ prod$

avec :



4.5.2 Curryfication

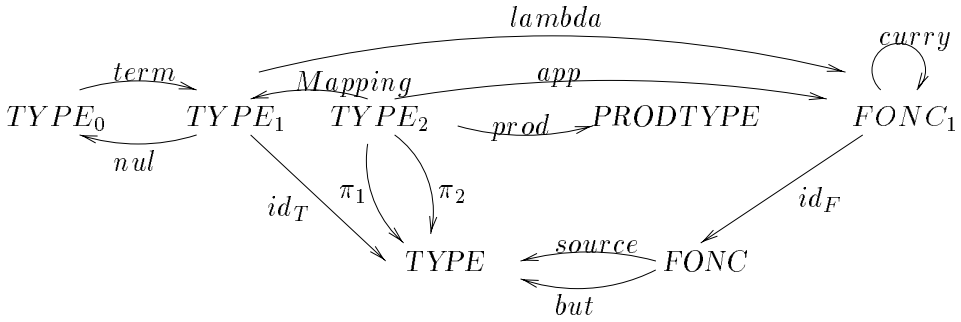
La possibilité qu'offre Aldor de curryfier toute fonction se traduit là aussi par la présence d'un constructeur de fonctions $curry$ dans $\mathbf{E}_{Aldor}$:



$$\begin{cases} source \circ curry = \ldots \\ but \circ curry = \ldots \end{cases}$$

4.5.3 Application

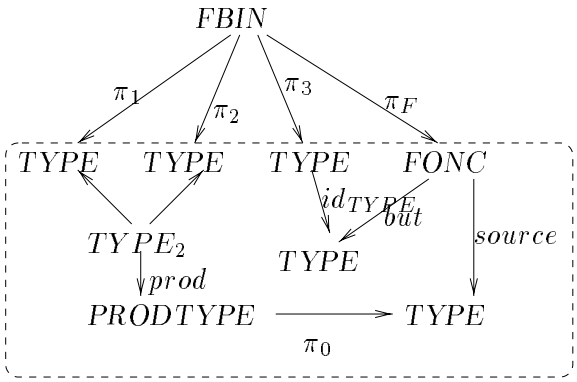
L'aspect fonctionnel du langage Aldor s'exprime au niveau du type des fonctions, puisqu'à chaque objet exponentiel, qui est un type, on associe une fonction d'application. Cette fonction d'application, produite par un constructeur noté *app* dans $\mathbf{E}_{Aldor}$, doit en outre vérifier une équation de compatibilité avec la composition induite de $\mathbf{E}_{Aldor}$:



On indique la relation entre application de fonction et existence d'une fonction d'application grâce à une équation supplémentaire, traduction au niveau esquisse de la propriété dernière du 3.8.2 :

$$comp \circ fact(f_1, f_2) = \pi_F$$

avec les objets et flèches suivants (*FBIN* désigne l'objet des fonctions d'arité 2) :



$\pi_{12} = fact(\pi_1, \pi_2)$, $\pi_{23} = fact(\pi_2, \pi_3)$, $fp_1 = \pi_1 \circ prod \circ \pi_{12}$, $fp_2 = \pi_2 \circ prod \circ \pi_{12}$

$$f_1 = app \circ \pi_{23}$$

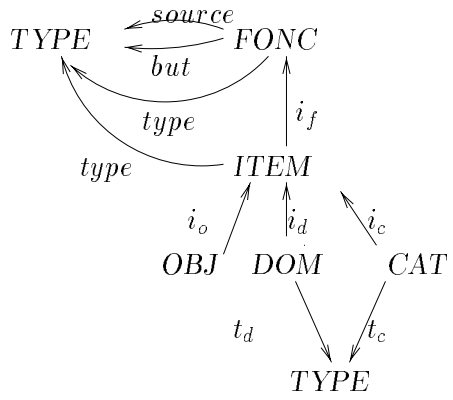
$$f_2 = factorise \circ fact(prod \circ fact(mapping \circ \pi_{23}, \pi_2), fact(comp \circ fact(curry \circ \pi_F, fp_1), fp_2))$$

Chapitre 5

Typage d'une expression par déclaration

5.1 Rappels et hypothèses

On rappelle la structure essentielle du chapitre précédent :



De même que l'on a une flèche *type* de *FONC* vers *TYPE*, il y a également une flèche *type* de *ITEM* vers *TYPE*, ce qui permet ainsi de définir le type de toute expression Aldor.

Le type d'une expression donnée n'est évidemment pas quelconque et la nature de l'expression détermine quels sont les catégories ou domaines utilisables pour typer cette expression. On différencie donc les types non

plus seulement en catégories et domaines, qui est une différence de nature, mais également en fonction de leur utilité, autrement dit en fonction des expressions dont ils peuvent être types.

On étudie dans ce chapitre la façon dont on détermine le type d'une expression identifiée, en supposant qu'une identification va de pair avec une déclaration de type.

On donne en toute fin les premiers cas de satisfaction de types, ceux ne concernant que les expressions identifiées et typées par déclaration.

5.1.1 Types de première espèce

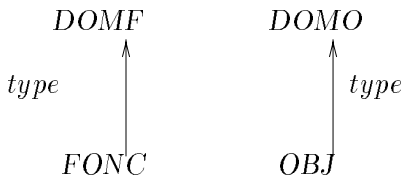
Toute fonction et tout objet de base se typent par un domaine, lequel domaine ne peut être ni `Type` ni `Category`(2.6). [1]

On parle alors de types de première espèce.

Le type d'une fonction est toujours un domaine de la forme $A \to B$, notons *DOMF* l'objet correspondant dans $\mathbf{E}_{Aldor}$ aux types des fonctions.

Le type d'un objet de base est un domaine qui n'est ni `Type`, ni `Category`, ni un type de fonction ; notons *DOMO* l'objet correspondant dans $\mathbf{E}_{Aldor}$.

On note *DOM1* l'objet des domaines qui sont distincts de `Type`, de `Category` et de tous les types qui sont satisfaits par `Type` et `Category`. Ainsi :



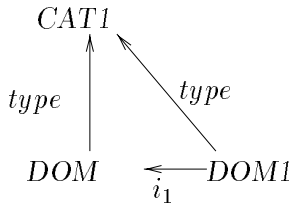
Les objets *DOM1*, *DOMO* et *DOMF* sont évidemment construits à partir de *DOM*, construction que l'on ne mentionne pas ici pour ne pas parasiter notre propos. On conserve juste une flèche i_1 :*DOM1*$\to$*DOM*.

Bien souvent, les flèches *type* seront composées avec les flèches d'injection de *DOM1*, *DOMO*, *DOMF* vers *DOM* puis vers *TYPE*, mais on gardera les mêmes noms de flèches pour faciliter la compréhension.

[1] Ni même `Cross(Type)`, `Tuple(Type)`, `Cross(Category)`, `Tuple(Category)`, et plus généralement les types qui sont satisfaits par `Type` et `Category`.

5.1.2 Types de deuxième espèce

Tous les domaines, y compris `Type` et `Category`, sont typables par les types dits de deuxième espèce que constituent les catégories et le domaine `Type`. On notera *CAT1* l'objet de ces types de deuxième espèce.

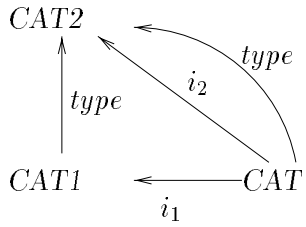


$$type \circ i_1 = type$$

Là encore *CAT1* se détermine à partir de *CAT*, construction dont on ne garde comme trace que la flèche i_1 : *CAT*→*CAT1*, avec l'intention de représenter par *CAT1* toutes les catégories ainsi que `Type`.

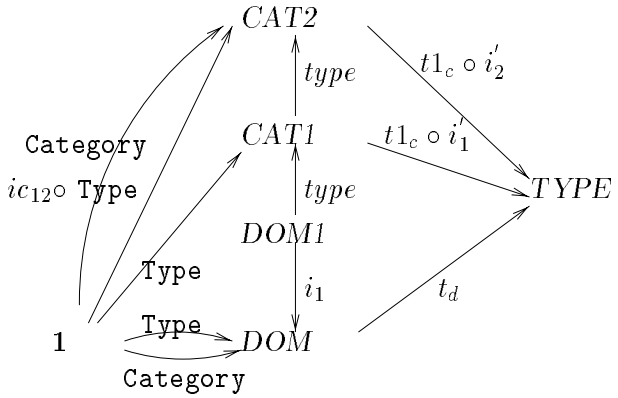
Les catégories sont elles-mêmes typables, par une catégorie ou par le domaine `Category` ou par le domaine `Type`.

Notons *CAT2* l'objet qui a vocation, dans une interprétation ensembliste, à représenter l'ensemble des catégories et les domaines `Category` et `Type`. On peut bien entendu esquisser dans $\mathbf{E}_{Aldor}$ cette spécification de *CAT2*.



$$type \circ i_1 = type$$

On a également *CAT1* qui s'envoie dans *CAT2* par $ic_{12} : CAT1 \to CAT2$. On utilise les types `Type` et `Category` comme de vrais domaines ou comme des types de domaines et de catégories, auquel cas on considère alors les flèches :

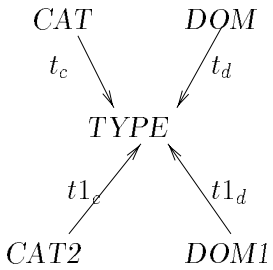


$$\begin{cases} t_d \circ \texttt{Type} &= t1_c \circ i_1' \circ \texttt{Type} \\ t_d \circ \texttt{Category} &= t1_c \circ i_2' \circ \texttt{Category} \end{cases}$$

Les flèches `Type` et `Category` utilisées dans cette section le seront aux compositions avec les injections près.

5.1.3 Présentations de *TYPE*

L'objet *TYPE* est donc présentable de deux manières :



5.1.4 Nature d'une expression

Une expression Aldor ne peut être que de nature :

1. fonction
2. objet de base
3. domaine

4. catégorie

et ceci de manière exclusive.

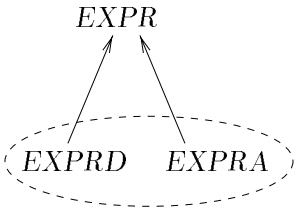
On fait donc figurer dans $\mathbf{E}_{Aldor}$ un nouvel objet NAT ainsi qu'une nouvelle flèche $nature$:

$$EXPR \overset{nature}{\longrightarrow} NAT$$

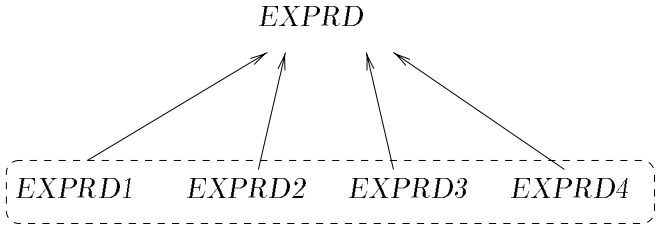
L'objet NAT se présente comme sommet d'un cône inductif distingué dans $\mathbf{E}_{Aldor}$ et un certain nombre d'équations donnent la valeur de la composée avec $nature$ pour des flèches de codomaine $EXPR$. On ne détaille pas ces éléments ici puisque seules des commutativités de diagrammes comportant l'objet NAT nous intéresseront par la suite.

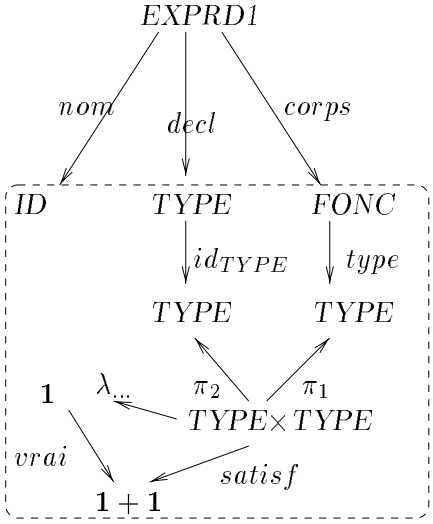
5.1.5 Expressions identifiées

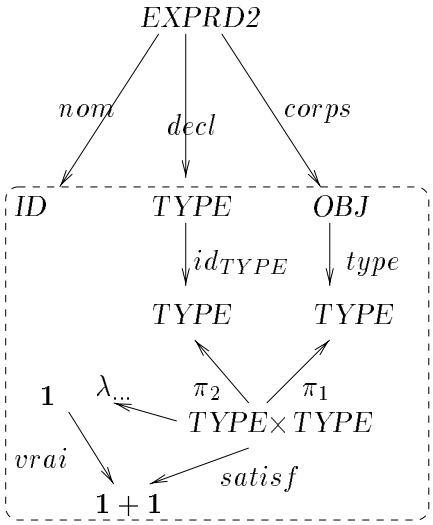
Parmi les expressions, on distinguera celles qui sont identifiées et typées par déclaration des expressions anonymes, d'où l'ajout d'un cône inductif distingué dans $\mathbf{E}_{Aldor}$:

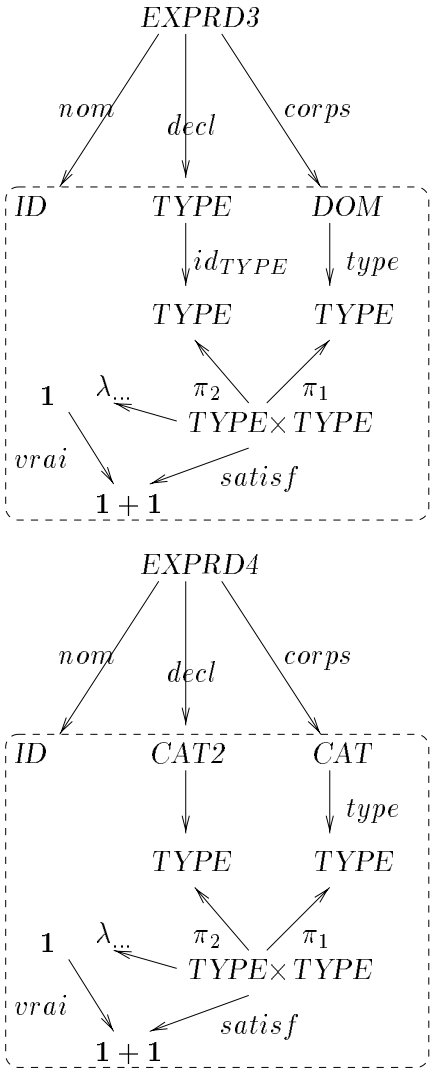


Les expressions identifiées et typées doivent être valides pour la déclaration, validité assurée par les cônes distingués suivants dans $\mathbf{E}_{Aldor}$, pour ce qui concerne uniquement la vérification d'une satisfaction de types :




EXPRD1
nom
decl
corps
ID
TYPE
FONC
id_{TYPE}
type
TYPE
TYPE
1
$\lambda_{...}$
π_2
π_1
TYPE×TYPE
vrai
satisf
1 + 1


EXPRD2
nom
decl
corps
ID
TYPE
OBJ
id_{TYPE}
type
TYPE
TYPE
1
$\lambda_{...}$
π_2
π_1
TYPE×TYPE
vrai
satisf
1 + 1



$$vrai \circ \lambda_{TYPE \times TYPE} = satisf$$

L'objet *ID* désigne la sorte des identificateurs d'Aldor, autrement dit toutes les chaînes de caratères à l'exception des quelques mots réservés du langage.

Hypothèse principale : On ne s'intéresse pour le moment qu'à la partie des expressions qui sont identifiées et typées par déclaration. Le cas des expressions anonymes fait l'objet du chapitre 6.

Tout type déclaré pour un item est une expression valide ; tout type déclaré pour une fonction est l'application de la fonction $\rightarrow$ à deux items qui sont des types.

On ne cherchera donc à spécifier que la flèche *type* de *EXPRD* vers *TYPE*, via les flèches *type* de *ITEM* (respectivement *FONC*) vers *TYPE*, elles-mêmes composées des flèches *type* de domaine ou codomaine *OBJ*, *DOM1*, *CAT1*, *CAT2*, ...

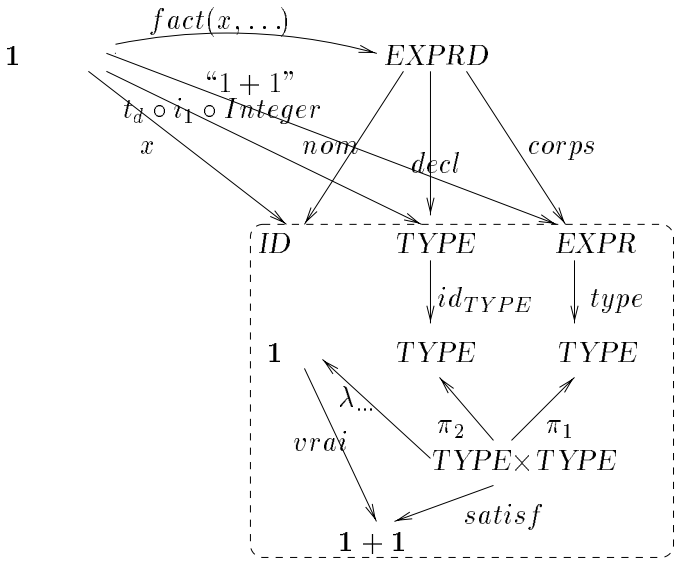
5.2 Type d'un item

Exemple 1 : `x : Integer == 1 + 1`

Cette instruction définit une expression identifiée et typée, où l'identificateur est `x`, le type déclaré le domaine `Integer` et le corps une application de fonction.

L'item ainsi défini est même un objet de base puisque son type déclaré n'est un type ni de domaine ni de catégorie. Sa nature se détermine en étudiant l'application de fonction qui le définit (cf 7.3).

Cette expression se traduit dans $\mathbf{E}_{Aldor}$ par :



$$type \circ fact(x, t_d \circ i_1 \circ Integer, \text{“}1 + 1\text{”}) = t_d \circ i_1 \circ Integer$$

où `Integer` : $\mathbf{1} \rightarrow DOM1$.

Exemple 2 : `d : BasicType == Integer add`$\{f(x : \%) : \% == x\ ;\}$
Cette définition de constante est celle d'un item d'identificateur `d`, de type déclaré la catégorie `BasicType`, et l'analyse seule du corps permet de déduire la nature de l'item, à savoir un domaine. Pareillement, $type \circ fact(d, t1_c \circ i_1' \circ BasicType, \ldots) = t1_c \circ i_1' \circ BasicType$, où `BasicType` : $\mathbf{1} \rightarrow CAT1$.

Exemple 3 : `c : Category == Group with`$\{g : \% \rightarrow \%\ ;\}$
Le domaine `Category` est le type déclaré de cette expression qui est une catégorie selon l'analyse du corps de l'expression :

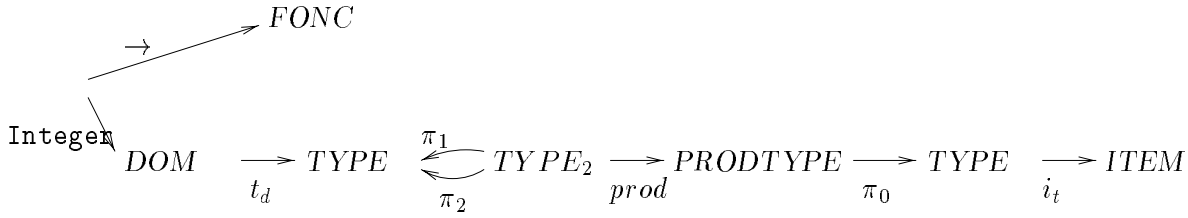
$$type \circ fact(c, t1_c \circ i_2' \circ Category, \ldots) = t1_c \circ i_2' \circ Category$$

Conclusion : Le type d'un item, lorsqu'il s'agit d'une expression identifiée et typée par déclaration, est exactement le type déclaré.

5.3 Type d'une fonction

Sous les hypothèses fortement simplificatrices de ce début de chapitre, une fonction identifiée et typée par déclaration est également une flèche de $\mathbf{1}$ vers $EXPRD$ dans $\mathbf{E}_{Aldor}$.

Exemple : `f : Integer→Integer == (`x`: Integer ): Integer +->` $x + 1$
L'identificateur de l'expression ainsi écrite en Aldor est `f`, le type déclaré est le domaine `Integer→Integer`. Il est obtenu par application de la fonction $\rightarrow$ au produit des domaines `Integer` et `Integer` : [2]



[2] L'application de $\rightarrow$ retourne un domaine.

Notons `(Integer, Integer)` la flèche :

$$i_t \circ \pi_0 \circ prod \circ fact(t_d \circ Integer, t_d \circ Integer)$$

Si l'on voulait composer cette flèche avec la flèche *type* de domaine *ITEM* et de codomaine *TYPE*, puis comparer le type obtenu avec celui désigné par la flèche $source \circ \rightarrow$, la relation de satisfaction serait vérifiée, ce que l'on ne peut établir pour l'instant puisque la relation de satisfaction sur les couples de types n'a pas encore été spécifiée.

Ainsi :

$$\mathbf{1} \xrightarrow{fac(id_{\mathbf{1}}, fact(\rightarrow, (Integer, Integer)))} FONCITEMCOMP \xrightarrow[itemcomp]{} DOM1 \xrightarrow[i_1]{} DOM$$

Le type de l'expression est alors donné par :

$$\begin{aligned} & type \circ fact(f, t_d \circ i_1 \circ itemcomp \circ fact(id_{\mathbf{1}}, fact(\rightarrow, (Integer, Integer))), \ldots) \\ = \;& t_d \circ i_1 \circ itemcomp \circ fact(id_{\mathbf{1}}, fact(\rightarrow, (Integer, Integer))) \end{aligned}$$

C'est là aussi le type déclaré pour la fonction.

5.4 Type d'une expression

A la lumière des deux paragraphes précédents, le type d'une expression identifiée et typée par déclaration est défini comme étant le type déclaré.

Définition 5.4.1 $type = decl$

5.5 Type et profil d'une fonction

Le type déclaré d'une fonction étant toujours de la forme $A \rightarrow B$ en Aldor, on parle alors de profil de la fonction, puisque A et B, respectivement le type source et le type but de la fonction, déterminent complètement ce type.

En outre, le type produit par l'application de $\rightarrow$ pouvant être comparé à un type exponentiel, on obtient donc la proposition suivante :

Proposition 5.5.1 *Considérons la flèche type* : $FONC \to TYPE$, *on a la relation*

$$type \;=\; mapping \circ profil$$

où $profil$ *est la flèche* $fact(source, but)$.

On démontre cette proposition en utilisant le résultat du chapitre précédent liant l'application de $\to$ au constructeur *mapping* (4.5.1) :

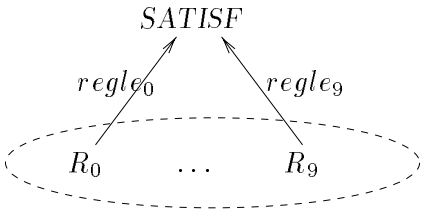
$$\begin{aligned} & type \\ &= t_d \circ itemcomp \circ fact(id_{\mathbf{1}} \circ \lambda_{ITEM}, fact(\to \circ \lambda_{ITEM}, id_{ITEM})) \\ &\quad \circ i_t \circ \pi_0 \circ prod \circ fact(source, but) \\ &= mapping \circ profil \end{aligned}$$

5.6 Relation de satisfaction

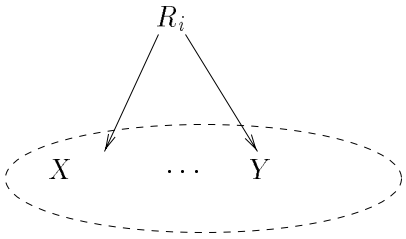
On s'intéresse de nouveau à l'objet *SATISF* de $\mathbf{E}_{Aldor}$ en donnant une nouvelle façon de réaliser une flèche de codomaine *SATISF*. Le principe est analogue à celui donné dans le chapitre précédent, à savoir spécifier *SATISF* comme sommet d'un cône distingué dans $\mathbf{E}_{Aldor}$.

L'objet *SATISF* est présenté dans le chapitre précédent (4.2.4) comme sommet d'un cône projectif distingué.

La relation de satisfaction sur les couples de types pouvant être définie par ailleurs par dix règles exclusives, on présente également l'objet *SATISF* comme sommet d'un cône inductif distingué dans $\mathbf{E}_{Aldor}$ cette fois, à base finie discrète :

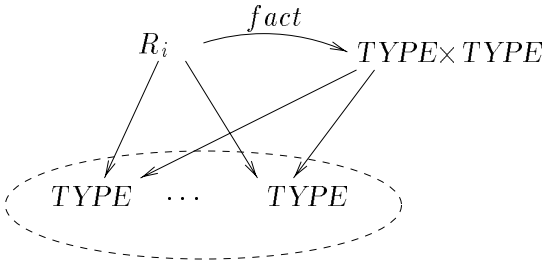


Pour $0 \leq i \leq 9$, chaque objet R_i est lui-même sommet d'un cône projectif distingué, de base un graphe fini :

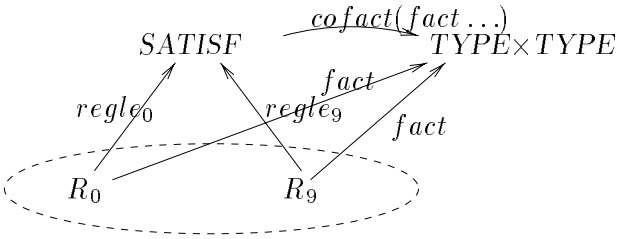


où les objets X et Y peuvent s'envoyer sur $TYPE$ par composition de flèches dans $\mathbf{E}_{Aldor}$, ils sont donc à prendre parmi les objets *TYPE*, *DOM*, *CAT*, *DOM1*, et coetera.

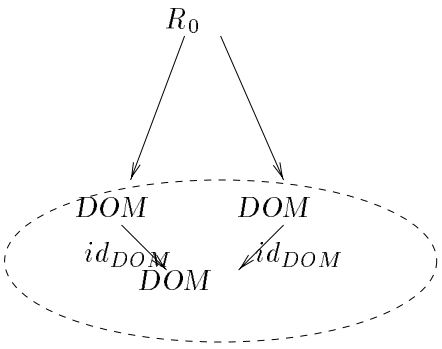
Dans $\mathbf{C}_{Aldor}$, ces cônes distingués deviennent des cônes limites et l'on dispose alors des flèches de factorisation et de cofactorisation vers les produits et les sommes :



Ainsi :

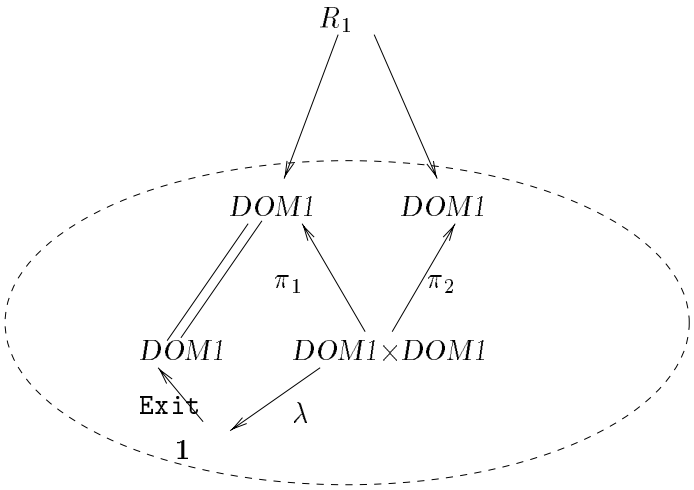


Règle 0 *Tout type domaine T satisfait T.*



Règle 1 *Le domaine* `Exit` *satisfait tout type* T *de première espèce.*

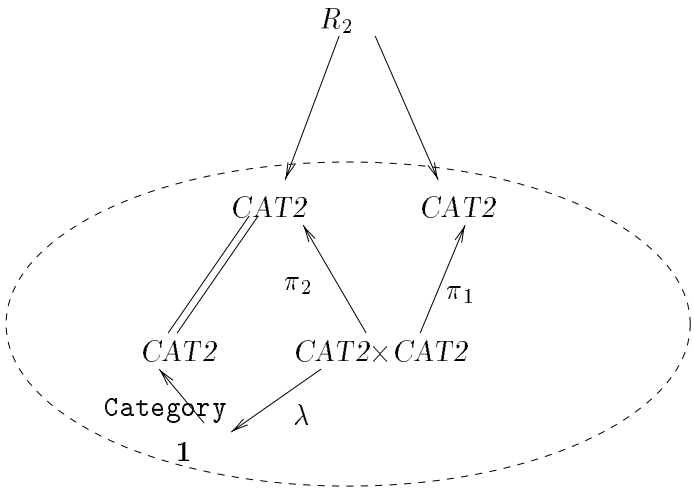
(`Exit` est un domaine prédéfini d'Aldor, d'intérêt limité pour ce qui nous intéresse)



$$\texttt{Exit} \circ \lambda_{DOM1 \times DOM1} = \pi_1$$

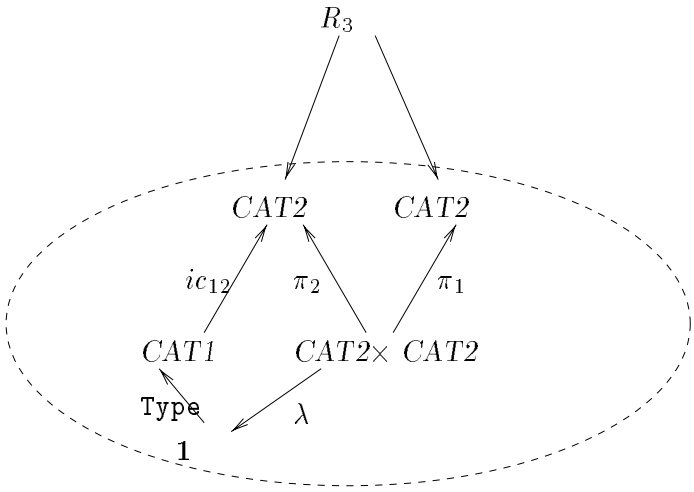
Règle 2 *Tout type de catégorie satisfait le type* `Category`.

(Pour mémoire, on a donc T satisfait `Category` s'il est "dans" $CAT2$)



$$\texttt{Category} \circ \lambda_{CAT2\times CAT2} = \pi_2$$

Règle 3 *Tout type T de domaine ou de catégorie satisfait le domaine* `Type`.



$$\texttt{Type} \circ \lambda_{CAT2\times CAT2} = \pi_2$$

Règle 4 *Tout sous-type d'un type T satisfait T.*

Initialement, dans Axiom,, un **sous-type**, relativement à un domaine de base fixé, est un type dont les constantes, variables ou expressions satisfont une propriété particulière. Ceci s'écrivait également :

si T_1 et T_2 sont deux sous-types ,relativement à un domaine de base D, et si tous les objets de type T_1 peuvent être déclarés également de type T_2, alors T_1 est un **sous-type** de T_2 et on note $T_1 \sqsubseteq T_2$.

Définissons maintenant une notion de sous-type en Aldor.

On définit la relation $\sqsubseteq$ **entre catégories identifiées** par :

Pour deux catégories identifiées C_1 et C_2, C_1 satisfait C_2 si et seulement si C_1 est construite explicitement (ie nommément) à partir de C_2.

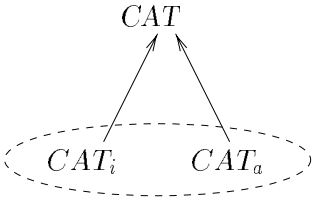
Exemple :

```
C2 :  Category == with {
        f  :  % → Integer ;
        +  :  (%,%) → % ;
        n  :  Boolean ;
}
```

```
C1 :  Category == C2 with {
        g  :  (%,%) → Boolean ;
}
```

La catégorie C_1 est construite à partie de C_2 et donc $C_1 \sqsubseteq C_2$.

Transcription dans $\mathbf{E}_{Aldor}$: Une catégorie en Aldor est soit identifiée soit anonyme, notons CAT_i l'objet des catégories identifiées et CAT_a l'objet des catégories anonymes, objets de la base d'un cône inductif distingué dans $\mathbf{E}_{Aldor}$:



On se restreint dans ce chapitre à CAT_i, les cas relatifs à CAT_a seront étudiés en 6.4.1.

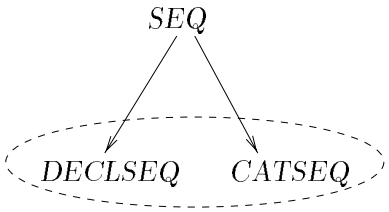
On a besoin d'exhiber la séquence d'exports d'une catégorie, notons *SEQ* l'objet correspondant dans $\mathbf{E}_{Aldor}$. Dans une telle séquence apparaissent des déclarations

`f : % → Integer`

et des identificateurs de catégories

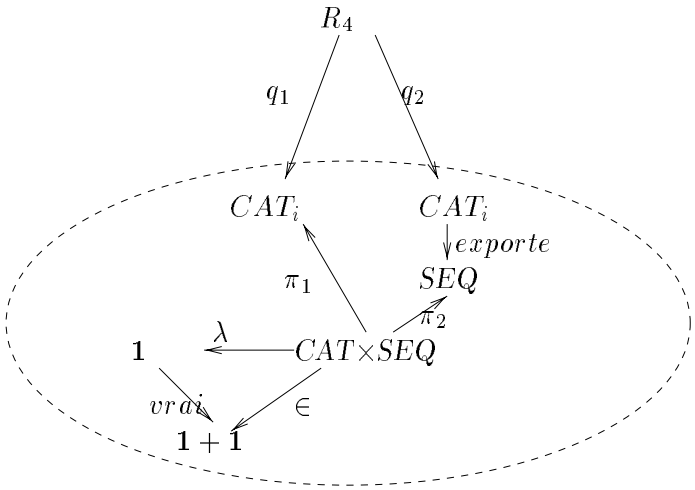
`C2`.

On présente donc *SEQ* comme sommet d'un cône projectif distingué dans $\mathbf{E}_{Aldor}$:



Le prédicat associé à la relation d'appartenance d'une catégorie identifiée à la séquence d'"exports" d'une autre se représente par une flèche $\in$ de $CAT \times SEQ$ vers $\mathbf{1}+\mathbf{1}$: `C2` appartient à la séquence d'exports de `C1`.

On est ainsi en mesure de spécifier R_4, objet des couples de catégories identifiées dont la première est dans la séquence d'"exports" de la seconde:



$$vrai \circ \lambda_{CAT \times SEQ} = \in$$

et ce cône de sommet R_4 est un cône projectif distingué dans $\mathbf{E}_{Aldor}$.

La description de la règle 4 pour les catégories anonymes se fera en 6.4.1.

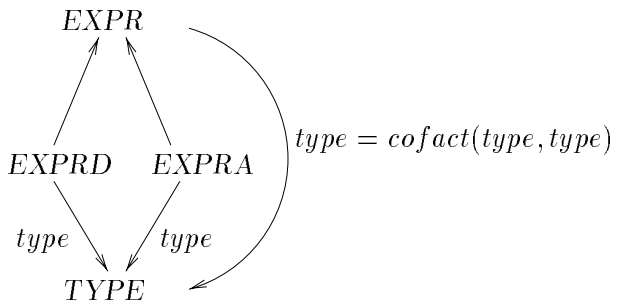
Chapitre 6

Construction et type inféré d'une expression

6.1 Hypothèses principales

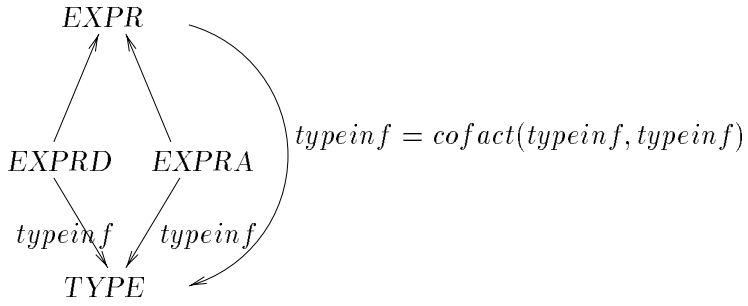
On rappelle que toute expression est soit anonyme soit identifiée et typée par déclaration, et que toute expression a un type.

La flèche *type* de domaine *EXPR* se détermine à partir des flèches de typage des expressions déclarées et des expressions anonymes. On se trouve donc dans la configuration suivante :



Cette description du typage n'est malheureusement pas "constructive" au sens où l'on n'a pas encore vu de quelle façon se construisent précisément les expressions et leurs types. C'est ce qui fait l'objet de ce chapitre avec tout d'abord l'introduction de la notion de type inféré.

Toute expression, qu'elle soit identifiée et typée par déclaration ou anonyme, a un type inféré :



Ainsi :

On a vu en 5.4 que pour une expression identifiée et typée par déclaration, le type est le type déclaré : $type = decl$.

La condition de validité pour une expression identifiée et typée par déclaration est : "*le type inféré satisfait le type déclaré*"

On définit le type inféré dans ce cas tout simple par :

$$typeinf = type \circ corps$$

Le type inféré est le type de l'expression qui définit l'expression identifiée.

Pour une expression anonyme, on a :

$$typeinf = type$$

On va dans ce chapitre spécifier plus généralement la flèche $typeinf$ de $EXPR$ vers $TYPE$. On le fera au cas par cas en étudiant les flèches $typeinf$ de $FONC$ vers $DOM1$, OBJ vers $DOM1$, DOM vers $CAT1$, CAT vers $CAT2$.

On commencera par distinguer les deux présentations possibles d'une expression, sous forme applicative ou sous forme primitive. Cette dernière présentation nous donnera l'occasion d'introduire un certain nombre de constructeurs primitifs d'expressions Aldor, donc de donner ainsi la façon de définir initialement des termes Aldor.

Dans la dernière partie, on traitera les cas restants de satisfaction de types, ceux faisant intervenir des expressions anonymes.

6.2 Expression sous forme applicative

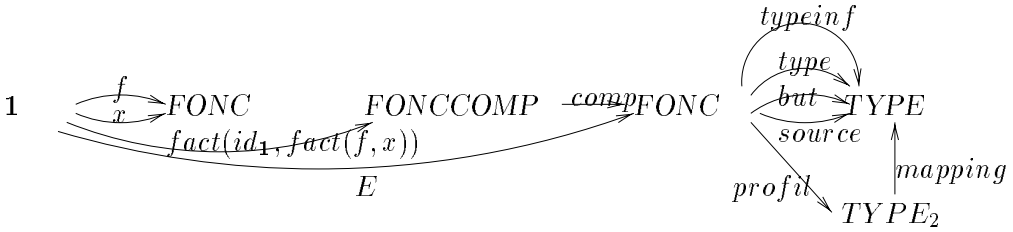
La forme applicative d'une expression correspond en Aldor à une expression de la forme $f(x)$ où f est une fonction et x une fonction ou un item.

Cette expression peut s'utiliser comme corps d'une expression identifiée et typée par déclaration ou comme expression anonyme. On la note E par commodité et on cherche à déterminer le type inféré de E.

E correspond dans $\mathbf{E}_{Aldor}$ à une flèche de $\mathbf{1}$ vers $EXPR$ qui se décompose en une composée de flèches.

6.2.1 Composée de fonctions

Si f et x sont des fonctions alors elles sont composables faiblement et :



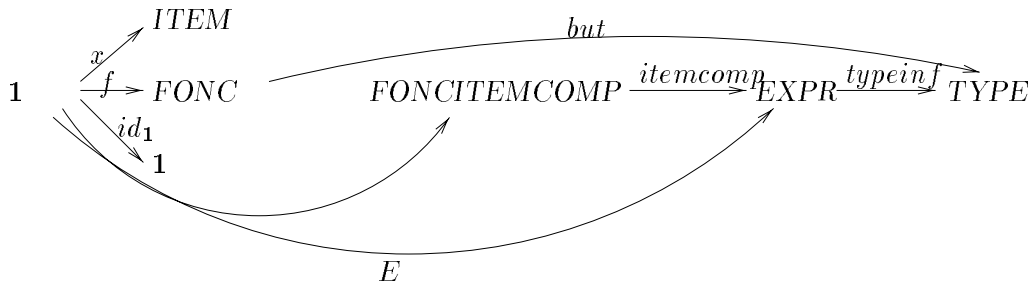
Le type inféré de la composée de fonctions est le type de la fonction composée. Les résultats des chapitres précédents avaient permis d'établir les relations $type = mapping \circ profil$ ainsi que de déterminer source et but de la fonction composée.

On obtient par conséquent :

$$typeinf \circ comp = mapping \circ fact(source \circ p_2 \circ q, but \circ p_1 \circ q).$$

6.2.2 Application d'une fonction à un item

Dans ce cas précis, x est un item et désigne donc un objet de base, un domaine ou une catégorie :



$$typeinf \circ itemcomp = but \circ p_1 \circ p_2$$

Remarque importante : Un cas problématique est celui d'une fonction dont le type-but est `Type`, ou un type que satisfait `Type`, ou une catégorie quelconque. A priori, on ne sait pas dans ce cas si l'expression retournée est un domaine ou une catégorie. Ceci ne peut se produire avec la façon dont nous définissons l'application de fonction en y introduisant la notion de nature (voir 7.3 pour vérification).

6.3 Expression sous forme primitive

Aldor dispose de primitives pour construire les fonctions, domaines et catégories : respectivement `+->`, `add` et `with`. On introduit ces primitives comme des constructeurs, représentés chacun dans $\mathbf{E}_{Aldor}$ par une flèche de codomaine *FONC*, *DOM* et *CAT* respectivement, et de domaine adéquat.

Les paragraphes suivants détaillent ces constructions et le typage associé.

6.3.1 Type inféré d'une fonction

On considère une fonction que l'on désigne par f, de définition :

```
(x : A):B +-> E
```

où `A` et `B` sont des types Aldor.

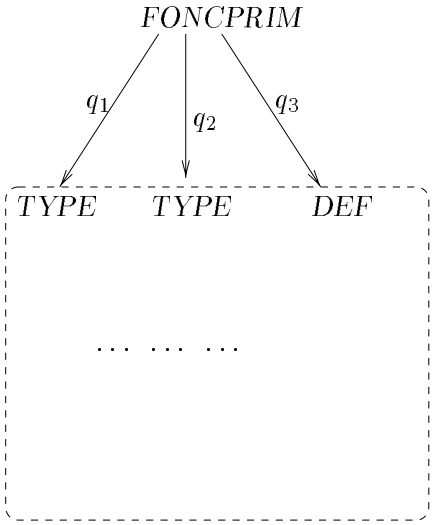
Alors le type inféré de f est le domaine `A` $\rightarrow$ `B`.

Traduisons cette propriété dans $\mathbf{E}_{Aldor}$.

On détermine tout d'abord le nouvel objet *FONCPRIM* domaine de la flèche `+->` de codomaine *FONC*.

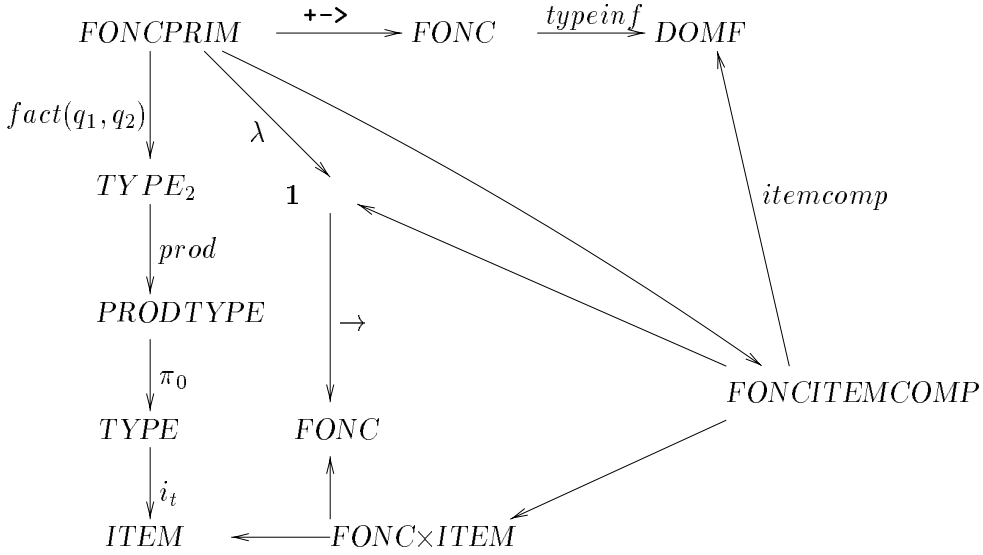
$$FONCPRIM \xrightarrow{\texttt{+->}} FONC$$

L'objet *FONCPRIM* est sommet d'un cône projectif distingué dans $\mathbf{E}_{Aldor}$:



On reviendra sur cet objet *DEF* dans le chapitre suivant qui détaille l'application de fonction (voir 7.3).

Pour définir le type inféré d'une fonction, on impose la commutativité au diagramme suivant :



$typeinf \circ \texttt{+->} \ = itemcomp \circ fact(\lambda_{FONCPRIM}, i_t \circ \pi_0 \circ prod \circ fact(q_1, q_2)))$

6.3.2 Type inféré d'une catégorie

Une forme primitive de catégorie se construit avec `with`,

Exemple : `with{ BasicType ; n : Integer ; loi :` $(\%, \%) \to \%\}$

à partir d'une séquence de catégories et de déclarations de fonctions, d'objets de base ou de types.

Dans $\mathbf{E}_{Aldor}$, `with` est une flèche de SEQ vers CAT :

$$SEQ \xrightarrow{\texttt{with}} CAT$$

$$typeinf \circ \texttt{with} \ = t_c \circ \texttt{with}$$

Remarque : Sur l'objet CAT_a des catégories anonymes, en se restreignant aux formes primitives, on a $typeinf = type$ et ce diagramme commute :

$$\begin{array}{ccc} CAT_a & \underset{type}{\overset{typeinf}{\rightrightarrows}} & TYPE \\ \downarrow{\scriptstyle id_{CAT_a}} & & \uparrow{\scriptstyle t_c} \\ CAT_a & \longrightarrow & CAT \end{array}$$

6.3.3 Type inféré d'un domaine

Une forme primitive de domaine se construit avec `add`,

Exemple : `HalfInteger add{ id(`$x : \%$`) :` $\% == x$ `}`

à partir d'un domaine "parent", ici `HalfInteger` et d'une séquence de définitions de constantes (fonctions, objets de base ou types).

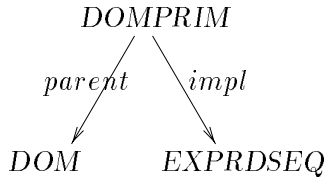
Dans l'exemple cité, le type inféré du domaine est la catégorie :

```
Join(with{coerce :  % →(HInt$Machine) ;
    coerce :  (HInt$Machine) → % },
Join(Logic, OrderedFinite, OrderedRing))
          with {id :  % → %}
```

On explicite la construction d'une forme primitive de domaine :

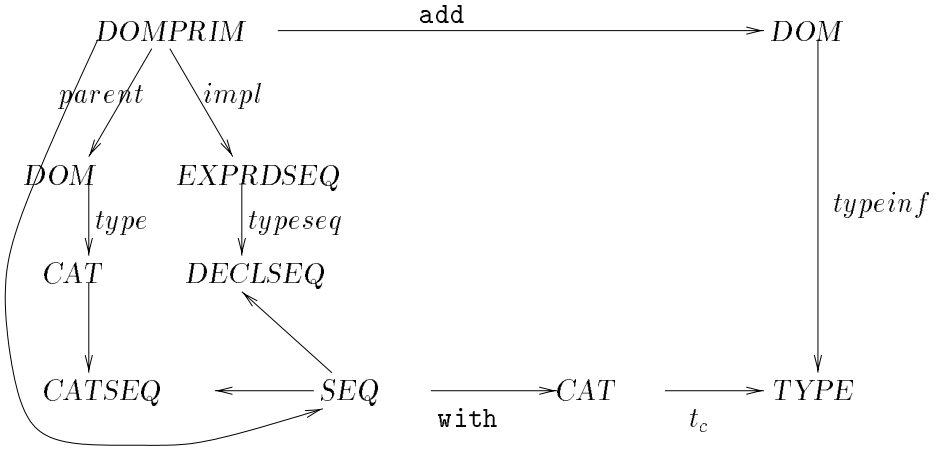
$$DOMPRIM \xrightarrow{\texttt{add}} DOM$$

en distinguant dans $\mathbf{E}_{Aldor}$ le cône projectif suivant :



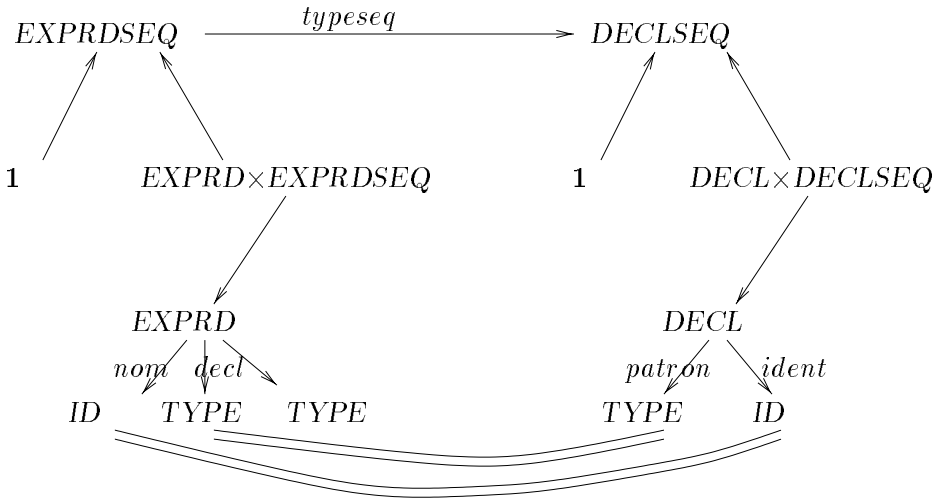
où *EXPRDSEQ* désigne l'objet des séquences d'expressions identifiées et typées par déclaration.

Pour déterminer le type inféré d'un domaine à partir d'une forme primitive :



$$typeinf \circ \texttt{add} = t_c \circ \texttt{with} \circ fact(\ldots)$$

où $fact$ est une flèche de factorisation de *DOMPRIM* vers *SEQ* dans $\mathbf{C}_{Aldor}$ et *typeseq* une flèche de *EXPRDSEQ* vers *DECLSEQ* que l'on décrit maintenant :



Remarque : On a utilisé dans cette construction le type du domaine "parent", qui est en général une catégorie sauf dans le cas où il se type en `type`. Dans cette situation très particulière, on considère alors le type inféré de la forme primitive initiale (voir ch. 7) du domaine "parent". Cette distinction de cas peut tout à fait se traduire dans le cadre de l'esquisse $\mathbf{E}_{Aldor}$ en ajoutant un cône inductif distingué.

6.4 Extensions de la relation de satisfaction

On traite dans cette section des cas de satisfaction de types non abordés au chapitre précédent, ceux-ci concernant des types non forcément identifiés.

6.4.1 Satisfaction de catégories

La règle générale est toujours la même (voir règle 4) :

Règle 5 *Tout sous-type d'un type T satisfait T,*

Elle était utilisée dans le contexte des catégories identifiées au 5.6, et l'on est maintenant en mesure de la compléter dans le contexte des expressions anonymes.

Pour deux expressions de catégorie C_1 et C_2, on note $Ex(C_1)$ et $Ex(C_2)$ l'ensemble des symboles exportés par chacune de ces deux catégories, et on a, relativement au domaine de base `Category` :

$$C_1 \sqsubseteq C_2 \Longleftrightarrow Ex(C_1) \supseteq Ex(C_2)$$

ainsi une expression de catégorie est un sous-type d'une autre si elle exporte plus d'opérations.

Exemple :

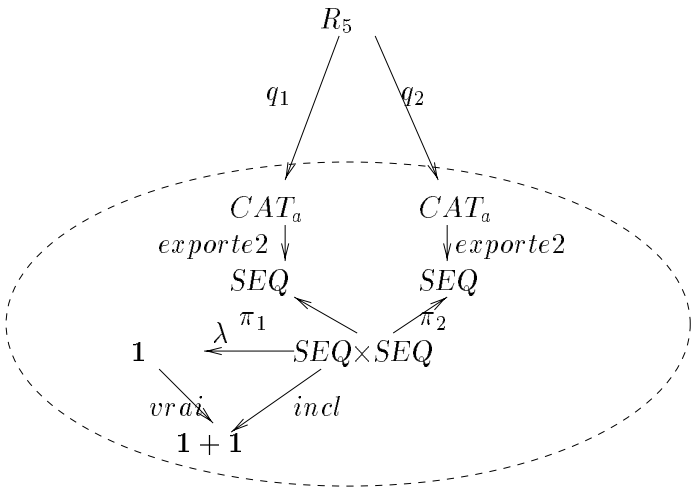
```
C2 ==> with {
       f  :  % → Integer ;
       +  :  (%,%) → % ;
       n  :  Boolean ;
}
```

```
C1 ==> with {
       f  :  % → Integer ;
       +  :  (%,%) → % ;
       n  :  Boolean ;
       g  :  (%,%) → Boolean ;
}
```

Alors $Ex(C_2) =<$ `f : %→Integer, + : (%,%)→ %, n : Boolean`$>$, et $Ex(C_1) =<$ `f : %→Integer, + : (%,%)→ %, n : Boolean, g : (%,%)→ Boolean`$>$.

On vérifie que $Ex(C_1) \supseteq Ex(C_2)$ donc l'expression de catégorie C_1 est un sous-type de l'expression de catégorie C_2.

Transcription dans $\mathbf{E}_{Aldor}$: On introduit pour ce nouveau cas de satisfaction l'objet R_5 des couples de **catégories anonymes** dont la première satisfait la seconde.



$$vrai \circ \lambda_{SEQ \times SEQ} = incl$$

La condition porte sur une vérification de l'inclusion des séquence d'exports. Elle n'est pas détaillée mais apparaît sous la forme d'une flèche de prédicat *incl*. Pour la définir rigoureusement dans $\mathbf{E}_{Aldor}$, il faudrait passer de la structure de séquences pour les exports à la structure d'ensembles afin de vérifier l'inclusion. Ce n'est pas l'objet de notre travail.

On définit également la satisfaction d'une **catégorie identifiée à une catégorie anonyme**, en considérant uniquement l'expression qui définit la catégorie identifiée, c'est une catégorie anonyme (si sous forme primitive mais on peut toujours s'y ramener).

Si C_1 est une catégorie identifiée et C_2 une expression de catégorie alors

$$C_1 \sqsubseteq C_2 \iff Ex(C_1) \supseteq Ex(C_2)$$

puisque l'on définit également la liste des "exports" d'une catégorie identifiée, c'est la liste des symboles exportés de l'expression qui la définit :

$$C_1 : \texttt{Category ==}\ E$$

où E est une expression de catégorie ; on pose $Ex(C_1) = Ex(E)$.
Exemple :

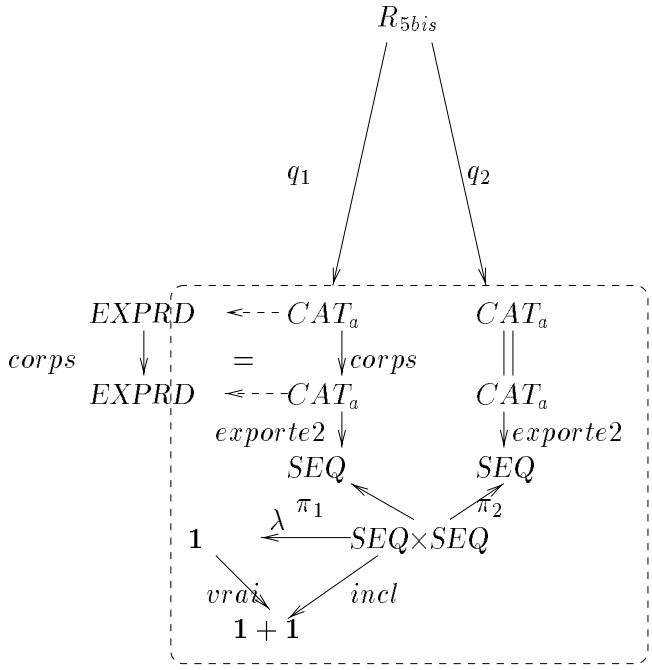
```
C2 ==> with {
       f  :  % → Integer ;
       +  :  (%,%) → % ;
       n  :  Boolean ;
}


C1 :  Category == with {
       f  :  % → Integer ;
       +  :  (%,%) → % ;
       n  :  Boolean ;
       g  :  (%,%) → Boolean ;
}
```

C_1 est une catégorie identifiée, ici une constante, et $Ex(C_1) =<$ `f : %→Integer`, `+ : (%,%)→ %`, `n : Boolean`, `g : (%,%)→ Boolean`$>$.

Donc $Ex(C_1) \supseteq Ex(C_2)$ et la catégorie C_1 est un sous-type de l'expression de catégorie C_2.

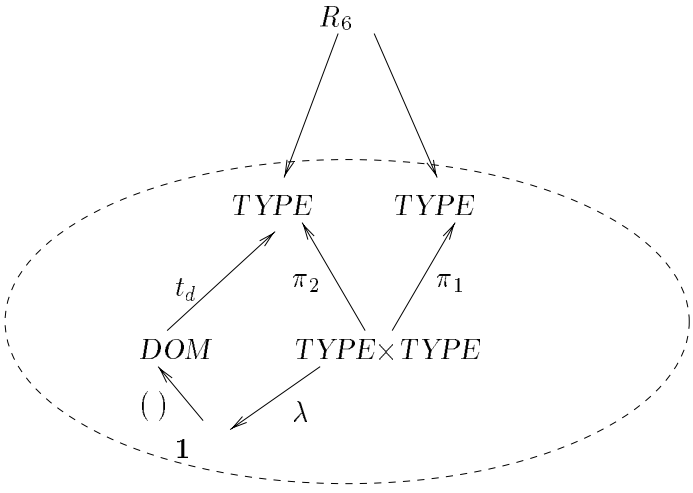
Transcription dans $\mathbf{E}_{Aldor}$: Ce nouveau cas est représenté par un nouvel objet R_{5bis} dans $\mathbf{E}_{Aldor}$, lui aussi sommet d'un cône projectif distingué :



$$vrai \circ \lambda_{SEQ \times SEQ} = incl$$

6.4.2 Satisfaction au type "vide"

Règle 6 *Tout type T satisfait le type $(\,)$.*



$$t_d \circ ( ) \circ \lambda_{TYPE \times TYPE} = \pi_2$$

6.4.3 Conversions automatiques

La règle générale est la suivante :

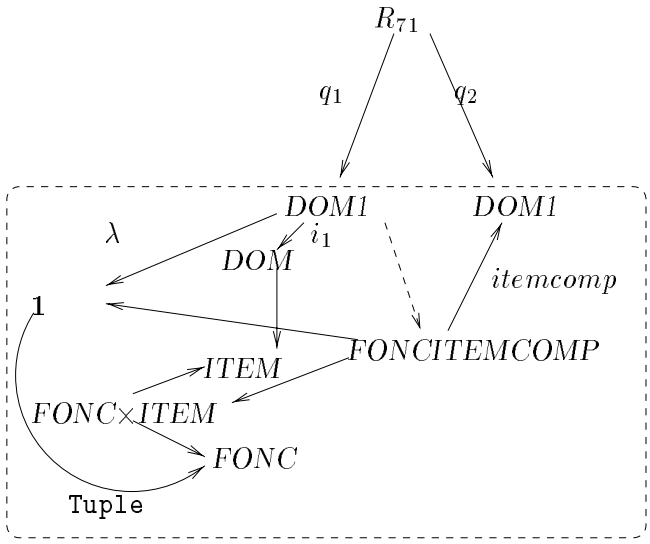
Règle 7 *Un type T_1 satisfait le type T_2 s'il existe une conversion "automatique" de T_1 vers T_2.*

On se contentera dans une première approche de donner trois cas de satisfaction issus de la règle générale.

Sous-règle 71 *Tout type de première espèce T satisfait le type de première espèce* `Tuple`(T).

La fonction `Tuple` d'Aldor s'applique à un argument dont le type satisfait `Type`. Traduite dans $\mathbf{E}_{Aldor}$, cette sentence exprime que T est une flèche de $\mathbf{1}$ vers *DOM1*.

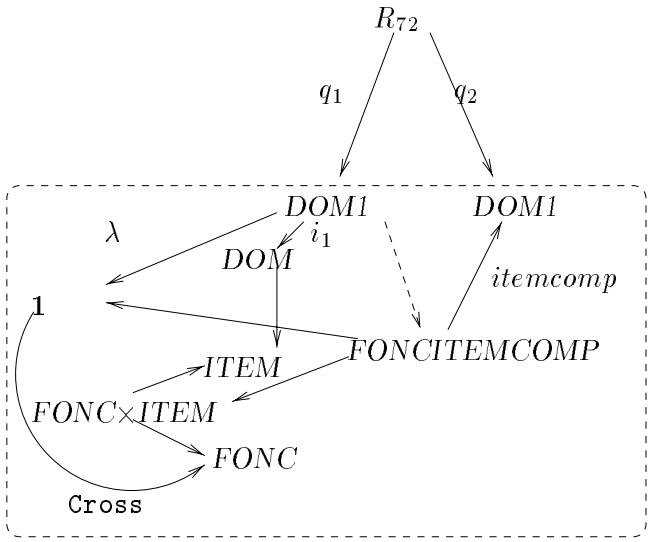
On considère alors le cône projectif distingué suivant dans $\mathbf{E}_{Aldor}$:



Sous-règle 72 *Tout type de première espèce T satisfait le type de première espèce* `Cross`(T).

La fonction `Cross` d'Aldor s'applique à un argument dont le type satisfait `Tuple(Type)`. Ceci signifie que T est représentable dans $\mathbf{E}_{Aldor}$ par une flèche de $\mathbf{1}$ vers *DOM1*.

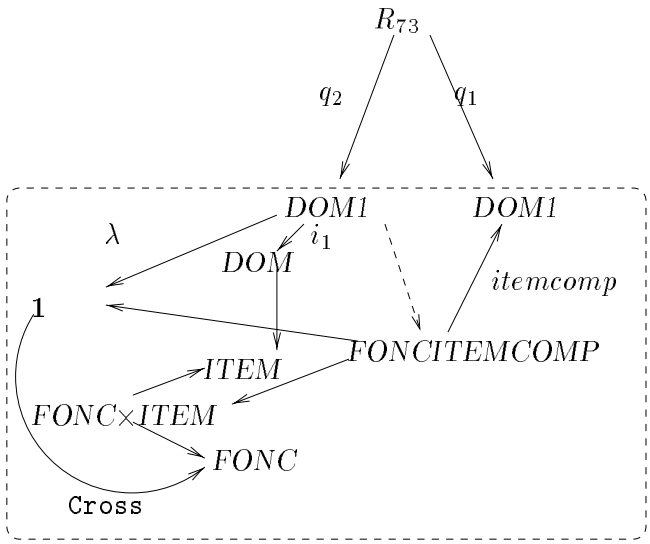
Ainsi cette deuxième sous-règle donne lieu à la construction suivante dans $\mathbf{E}_{Aldor}$:



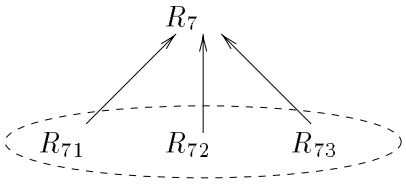
et ce cône projectif est distingué dans $\mathbf{E}_{Aldor}$.

Sous-règle 73 *Tout type de première espèce* `Cross`(T) *satisfait le type de première espèce* T.

On a donc :



L'objet représentatif de cette règle de satisfaction de types est l'objet R_7, sommet d'un cône inductif distingué dans $\mathbf{E}_{Aldor}$:

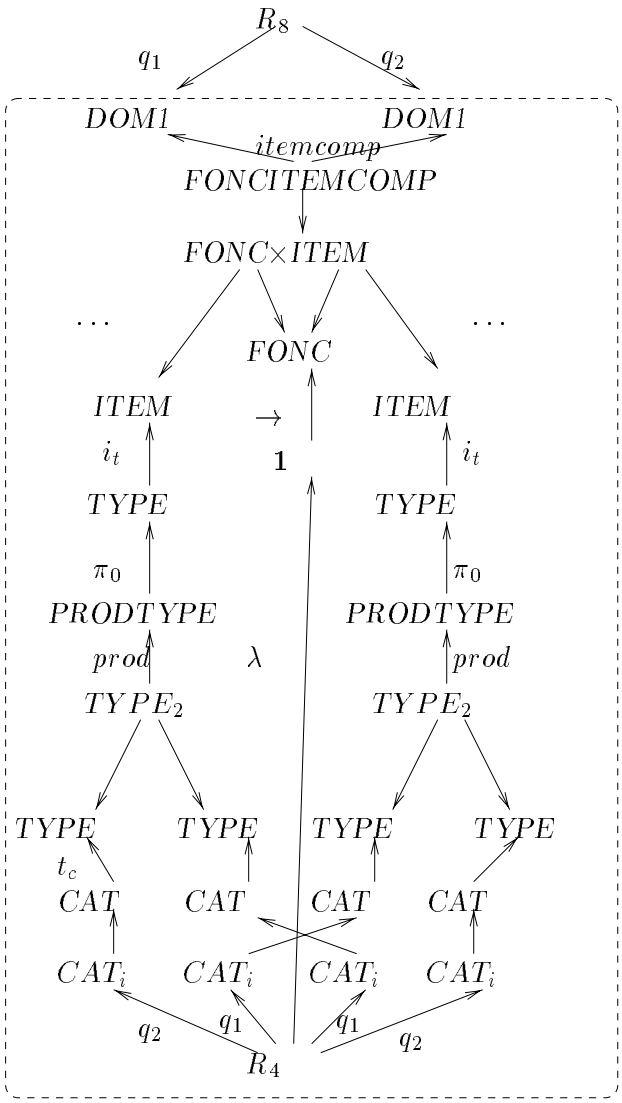


6.4.4 Compatibilité avec $\rightarrow$

La relation de satisfaction est compatible avec l'application de la fonction $\rightarrow$ d'Aldor aux sous-catégories.

Règle 8 *Soient deux couples de catégories identifiées (S_1, S_2) et (T_1, T_2), où S_2 satisfait S_1 et T_1 satisfait T_2, alors $S_1 \rightarrow T_1$ satisfait $S_2 \rightarrow T_2$.*

On introduit un nouvel objet R_8 de $\mathbf{E}_{Aldor}$ pour figurer cette satisfaction de types, toujours comme sommet d'un cône projectif distingué dans $\mathbf{E}_{Aldor}$, et cet objet est à ajouter à la somme des configurations de satisfaction dans $\mathbf{E}_{Aldor}$ pour définir convenablement *SATISF* :



Dans cette présentation on a considéré des catégories identifiées, mais en toute généralité S_1, $S_2, \ldots$ ne sont pas forcément des catégories identifiées mais peuvent être de la forme $S_1 = S_1' \rightarrow T_1'$, etc. Il faudrait alors décrire cette règle sous une forme récursive, ce qu'il est tout à fait possible de faire mais cela n'ajouterait pas à la clarté du propos. Nous nous en dispensons donc.

La réciproque est également vraie, si $S_1 \rightarrow T_1$ satisfait $S_2 \rightarrow T_2$ alors S_2 satisfait S_1 et T_1 satisfait T_2.

6.4.5 Satisfaction induite par `Join`

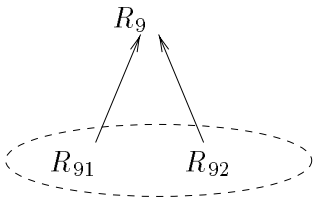
Parmi les fonctions prédéfinies d'Aldor, à l'instar de $\rightarrow$, `Tuple, Cross`, il existe une relation de satisfaction particulière pour la fonction `Join` :

```
Join (A : Category, B : Category) :  Category == with
```

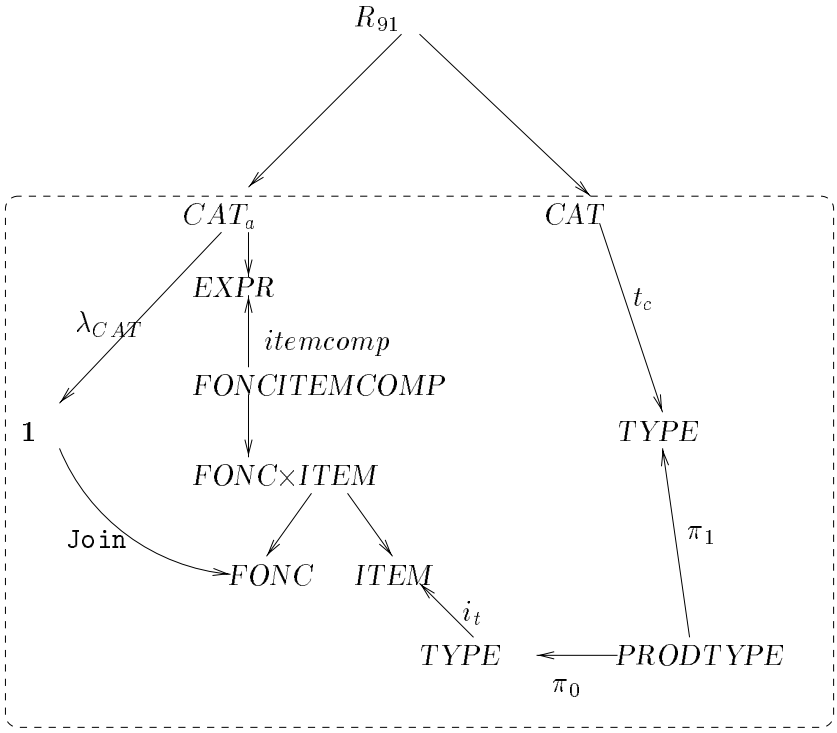
Bien qu'il n'y ait rien dans le corps de la fonction qui indique un héritage quelconque de catégories puisque la séquence d'exports est a priori vide, une catégorie créée par l'application de `Join` a pour exports la réunion des exports des catégories en arguments. On obtient donc un cas de satisfaction supplémentaire :

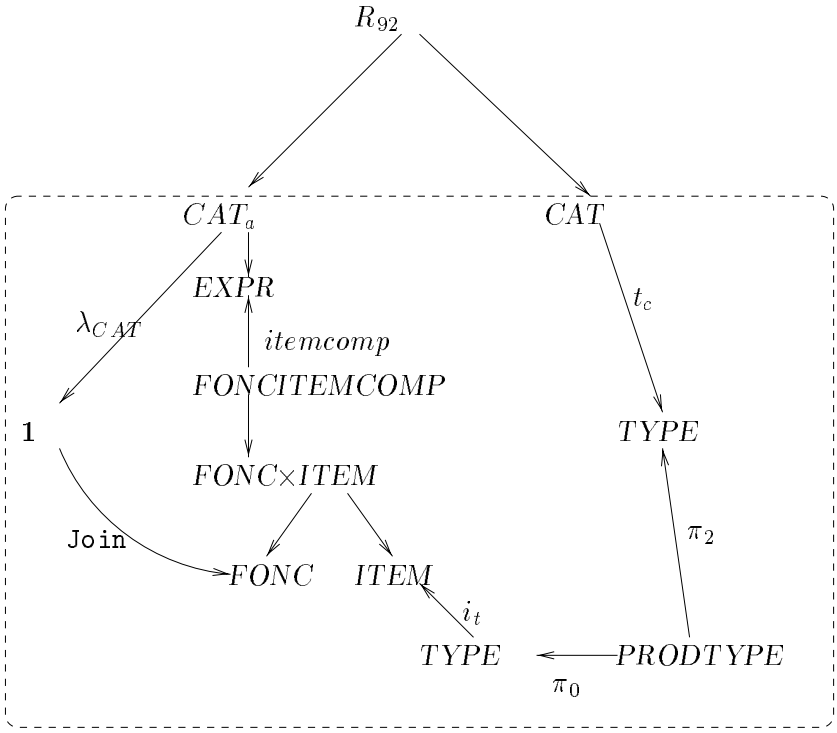
Règle 9 *Si C_1 et C_2 sont deux catégories, alors la catégorie* `Join`(C_1, C_2) *satisfait C_1 et C_2.*

Notons R_9 l'objet représentatif de ce dernier cas de satisfaction de types, il est sommet d'un cône inductif distingué dans $\mathbf{E}_{Aldor}$:



où R_{91} et R_{92} sont eux-mêmes sommets de cônes projectifs distingués dans $\mathbf{E}_{Aldor}$:


R_{91}
CAT_a
CAT
$EXPR$
λ_{CAT}
$itemcomp$
t_c
$FONCITEMCOMP$
$\mathbf{1}$
$TYPE$
$FONC{\times}ITEM$
Join
π_1
$FONC$
$ITEM$
i_t
$TYPE$
$PRODTYPE$
π_0


R_{92}
CAT_a
CAT
$EXPR$
λ_{CAT}
$itemcomp$
t_c
$FONCITEMCOMP$
$\mathbf{1}$
$TYPE$
$FONC{\times}ITEM$
Join
π_2
$FONC$
$ITEM$
i_t
$TYPE$
$PRODTYPE$
π_0

Chapitre 7

Application, héritage et appartenance

7.1 Introduction

On revient dans ce chapitre sur des spécificités du langage Aldor, abordables après avoir défini le type de toute expression.

On a vu notamment au cours des chapitres précédents que dans un contexte

```
x : T
```

où `T` est `Type` ou une catégorie quelconque, a priori une instance de `x` pourrait être un domaine ou une catégorie si on ne fait appel qu'à une vérification de satisfaction de types.

On montre ici, surtout dans un contexte d'application de fonction puisque c'est là que les vérifications sont les plus fortes en Aldor, que la notion de nature permet de lever cette ambiguité.

7.2 Compatibilité entre type et nature

Toute expression du langage Aldor possède un type et une nature.

Dans un contexte de déclaration `x : T`, on doit s'assurer évidemment que le type d'une instance de `x` est en relation de satisfaction avec `T`.

Mais dans un contexte `x : T == E`, on doit en outre s'assurer que le type requis pour une instance de `x`, ici `T`, est compatible avec la nature requise pour une instance de `x` par l'expression `E`.

Exemple : Considérons la fonction

`f : Category` $\rightarrow$ `Category == (c:Category):Category +-> c with {`
`g :`% $\rightarrow$ %`}`

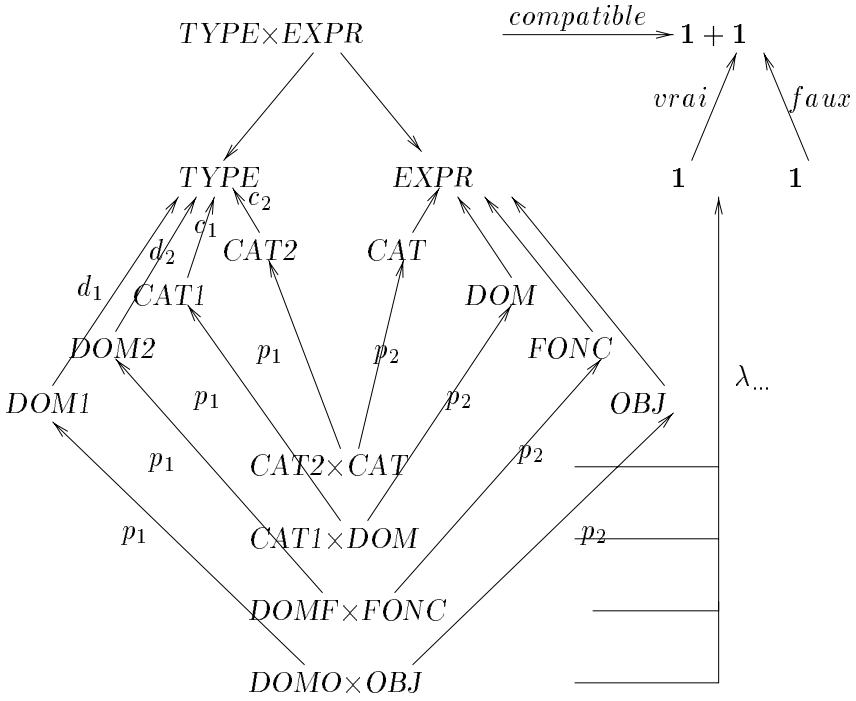
`c` doit être de nature une catégorie (à cause du constructeur de catégories `with`)et donc de type un type qui satisfait `Category`.

Or le type `Category` de la déclaration est compatible avec toute expression instance de `c` (qui est une catégorie) puisque le type `Category` peut typer une catégorie, donc le typage est satisfaisant (idem pour le type de sortie de la fonction).

On traduit cette compatibilité par l'ajout d'une flèche *compatible* dans $\mathbf{E}_{Aldor}$:

$$TYPE{\times}EXPR \xrightarrow{compatible} \mathbf{1}+\mathbf{1}$$

définie par les équations :



$$compatible \circ fact(d_1 \circ p_1, e_i \circ i_o \circ p_2) = vrai \circ \lambda_{DOMO \times OBJ}$$

$$compatible \circ fact(d_2 \circ p_1, e_f \circ p_2) = vrai \circ \lambda_{DOMF \times FONC}$$

$$compatible \circ fact(c_1 \circ p_1, e_i \circ i_d \circ p_2) = vrai \circ \lambda_{CAT1 \times DOM}$$

$$compatible \circ fact(c_2 \circ p_1, e_i \circ i_c \circ p_2) = vrai \circ \lambda_{CAT2 \times CAT}$$

et par complémentarité on obtient les équations qui définissent la non-satisfaction de la relation de compatibilité.

7.3 Retour sur la définition de fonction

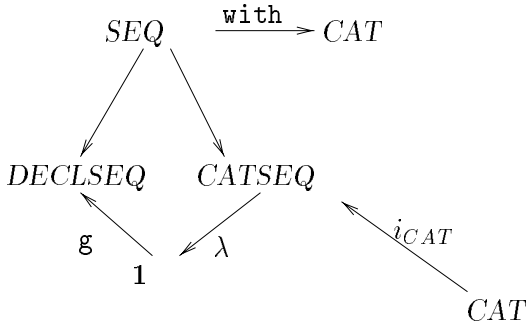
Exemple : Considérons la fonction

```
f : Category → Category == (c:Category):Category +-> c with {
                                    g :% → %}
```

représentable dans $\mathbf{C}_{Aldor}$ par une flèche de $\mathbf{1}$ vers $FONC$. L'identificateur `f` et le type `Category → Category` sont eux représentables par des flèches de $\mathbf{1}$ vers ID et vers $TYPE$ respectivement.

Le terme de droite définissant la fonction est une forme primitive de fonction, utilisant la primitive `+->`, on s'intéresse à traduire le "terme paramétré" qui détermine le corps de la fonction. Comme terme ouvert, autrement dit avec variable dans une écriture classique, c'est a priori une flèche de *EXPR* vers *EXPR* dans $\mathbf{C}_{Aldor}$.

On peut même préciser qu'il s'agit ici d'une flèche de *CAT* vers *CAT* uniquement d'après le typage, ce qu'il n'est pas toujours possible de faire.



Le terme ouvert correspond à la flèche

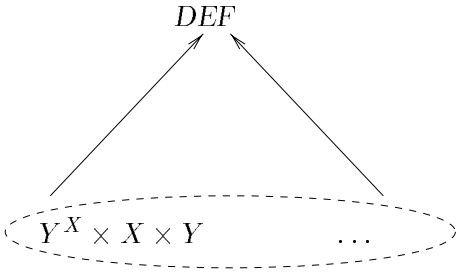
$$\texttt{with} \circ \mathit{fact}(\texttt{g} \circ \lambda_{CATSEQ} \circ i_{CAT}, i_{CAT})$$

de *CAT* vers *CAT*.

Il reste à intégrer cette flèche dans la définition d'une fonction telle qu'elle est présentée par l'objet *FONCPRIM* et le cône projectif distingué qui lui est associé. Précisément il s'agit de former une flèche de **1** vers *DEF* pour obtenir le corps d'une fonction.

Afin d'assurer l'existence d'une telle flèche, la propriété utilisée est celle d'esquisse généralisée, avec la construction des objets exponentiels et les mécanismes de curryfication et d'application.

On précise alors la base du cône inductif distingué de sommet *DEF* dans $\mathbf{E}_{Aldor}$ par un graphe fini discret :



où les objets X et Y sont à prendre parmi *OBJ, DOM, CAT* et *FONC*.

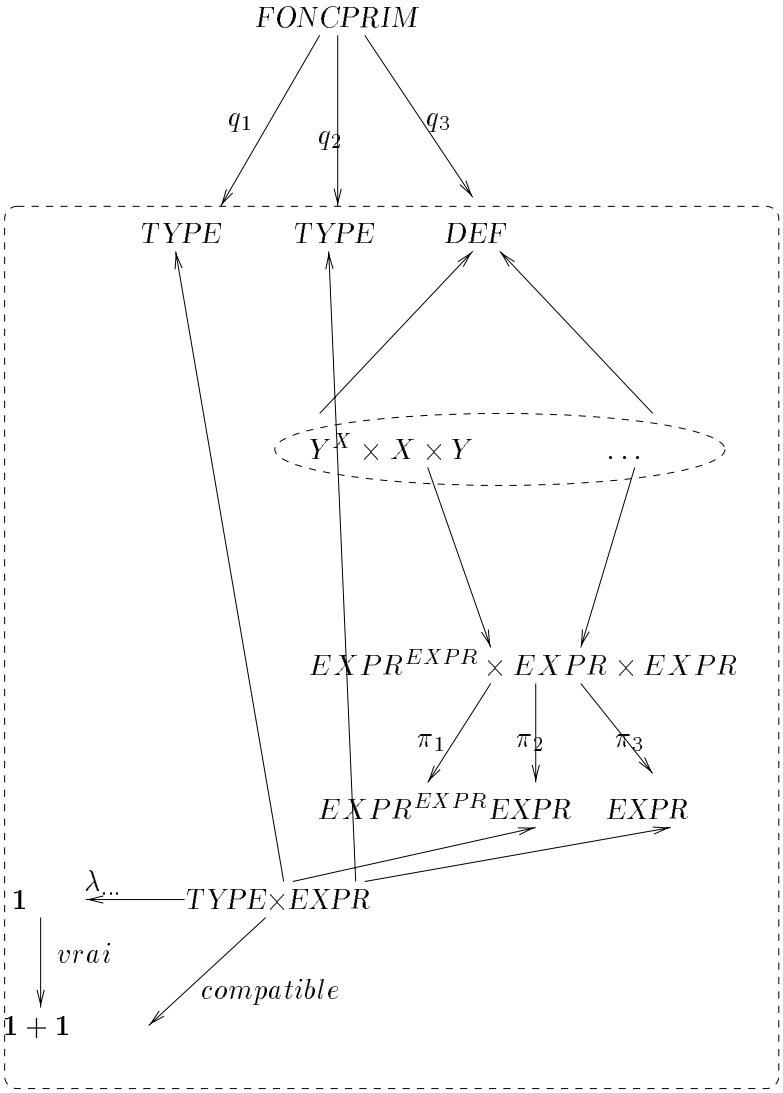
Si l'on note $fparam$ la flèche

$$\text{with} \circ fact(g \circ \lambda_{CATSEQ} \circ i_{CAT}, i_{CAT})$$

alors on a également dans $\mathbf{C}_{Aldor}$ sa curryfiée $fparamc$ de domaine $\mathbf{1}$ et codomaine CAT^{CAT}.

On obtient bien ainsi une flèche de $\mathbf{1}$ vers *DEF*.

Il reste à ajouter dans la base de *FONCPRIM* les conditions qui font de cette définition une définition de primitive de fonction valide, en termes de compatibilité notamment.



$$compatible = vrai \circ \lambda_{TYPE \times EXPR}$$

Ayant maintenant défini une fonction, ie une flèche de **1** vers *FONC* en composant avec `+->`, on montre que la condition requise pour $\mathbf{C}_{Aldor}$ permet de déterminer complètement la flèche *itemcomp* de *FONCITEMCOMP* vers *EXPR*, introduite au paragraphe 4.3.

7.4 Application d'une fonction à un item

On revient dans ce paragraphe sur la définition de l'application de fonction, amorcée auparavant avec la définition du cône distingué de sommet *FONCITEMCOMP*.

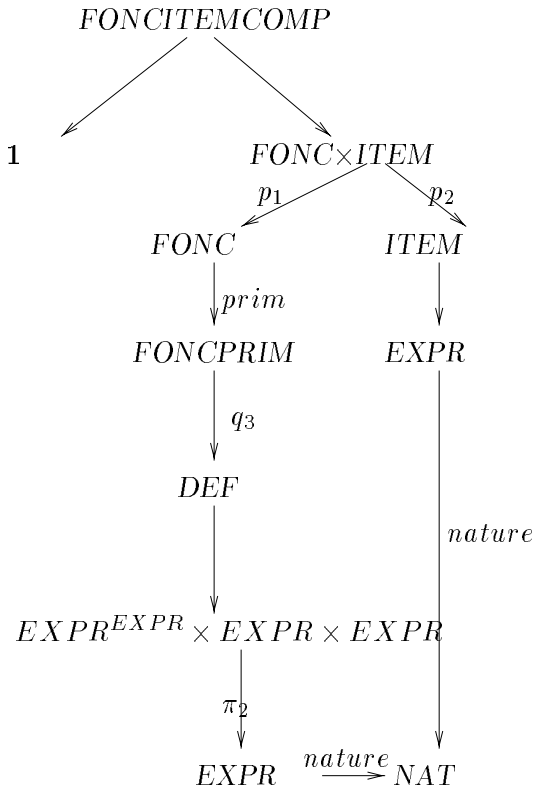
Pour parler d'application, on se donnait une fonction et un item dans une configuration de vérification de types.

Il s'agit maintenant d'évoquer le passage des paramètres, où l'on s'aperçoit que la vérification de la relation de satisfaction sur les types est insuffisante pour assurer la validité d'une application.

7.4.1 Première condition

Reprenons la fonction `f` de l'exemple précédent, la flèche associée $fparam$ est une flèche de CAT vers CAT. Elle représente une expression contextuelle et sa composée avec toute flèche de $\mathbf{1}$ vers CAT représente alors une catégorie d'Aldor.

Il s'avère donc nécessaire pour composer et obtenir une expression d'Aldor, de conditionner initialement la nature de l'argument, à savoir une catégorie dans le cas présent. Cette condition exprime que la nature de l'argument éventuel doit être identique à celle requise pour la définition de fonction, autrement dit le diagramme ci-dessous est commutatif :



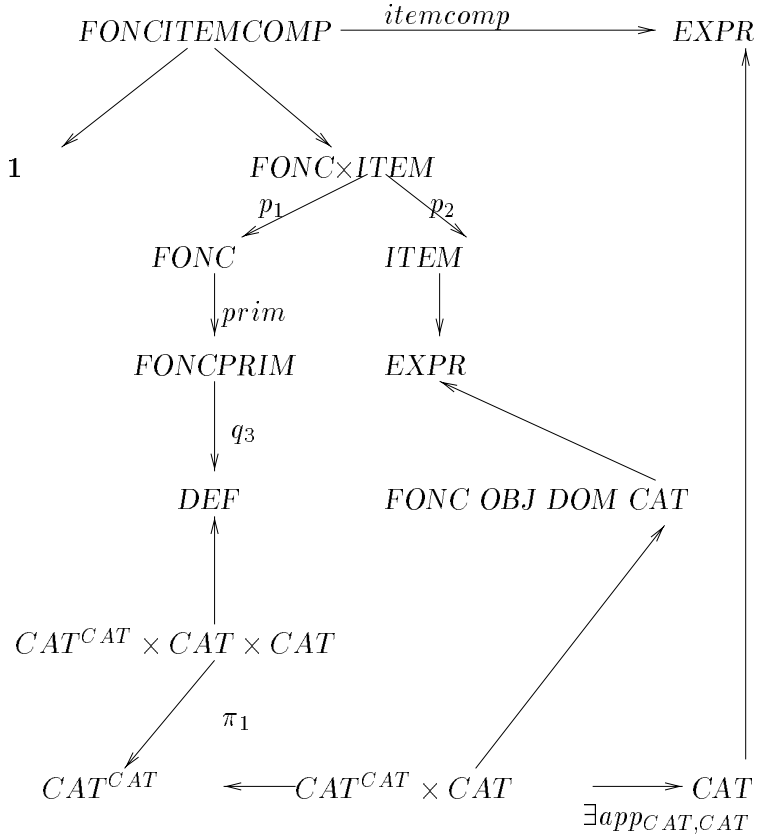
7.4.2 Seconde condition

La composition étant rendue possible au sens catégorique, il faut s'assurer qu'elle l'est également au sens d'Aldor, ie à une satisfaction de types près. Cette condition est obtenue par la structure de catégorie à composition faible définie au chapitre 4.

7.4.3 Justification de l'application

Dans la définition d'une catégorie à composition faible, une flèche *itemcomp* de *FONCITEMCOMP* vers *EXPR* traduit l'obtention d'une expression à partir d'une fonction et d'un argument. On peut maintenant justifier que l'on dispose naturellement d'une telle flèche. Elle provient de la structure d'esquisse généralisée imposée à $\mathbf{E}_{Aldor}$, où l'on dispose nturellement des flèches d'application de $B^A \times A$ vers B, pour tous objets A et B.

Démonstration par l'exemple (toujours avec la fonction `f`) dans $\mathbf{C}_{Aldor}$:



Pour une démonstration complète, les flèches $app_{CAT,CAT}$, $app_{CAT,DOM}$, etc liées à chacun des objets $Y^X \times X \times Y$ dans la base du cône issu de *DEF*, permettent de construire un cône de base discrète *FONC,OBJ, DOM, CAT*, ie de même base que le cône issu de*EXPR*. On obtient ainsi par factorisation et cofactorisation dans $\mathbf{C}_{Aldor}$ une flèche de *FONCITEMCOMP* vers *EXPR*, celle que l'on avait notée auparavant *itemcomp*.

Il est à noter que la condition sur le type du résultat de l'application de fonction a été donnée au chapitre 4, conjointement à celle sur le type de l'argument.

7.5 Perspectives

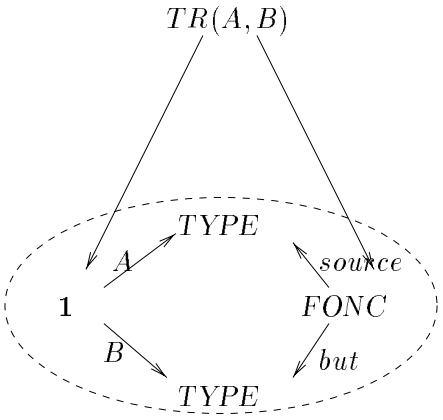
7.5.1 Autre point de vue sur l'application et la composition

On se propose ici de donner une présentation légèrement différente de la composition de fonctions. Par constructions dans $\mathbf{C}_{Aldor}$, en utilisant uniquement les propriétés des cônes limites, on va présenter la composition de fonctions comme une composée de flèches consécutives dans $\mathbf{C}_{Aldor}$ (au sens catégorique).

Il est à noter que cette présentation de la composition englobe l'application d'une fonction à un item puisque l'on a vu que, grâce à la flèche i_f, un item peut se concevoir comme une fonction.

Notion de "trace"

Etant donnés deux types A et B, ie deux flèches de $\mathbf{1}$ vers $TYPE$, on a dans $\mathbf{C}_{Aldor}$ la flèche de factorisation correspondante vers $TYPE_2$, notons la (A, B). A cette flèche de $\mathbf{1}$ vers $TYPE_2$, on associe un objet, appelé "trace" de A et B et noté $TR(A, B)$, sommet d'un cône limite projective dans $\mathbf{C}_{Aldor}$:

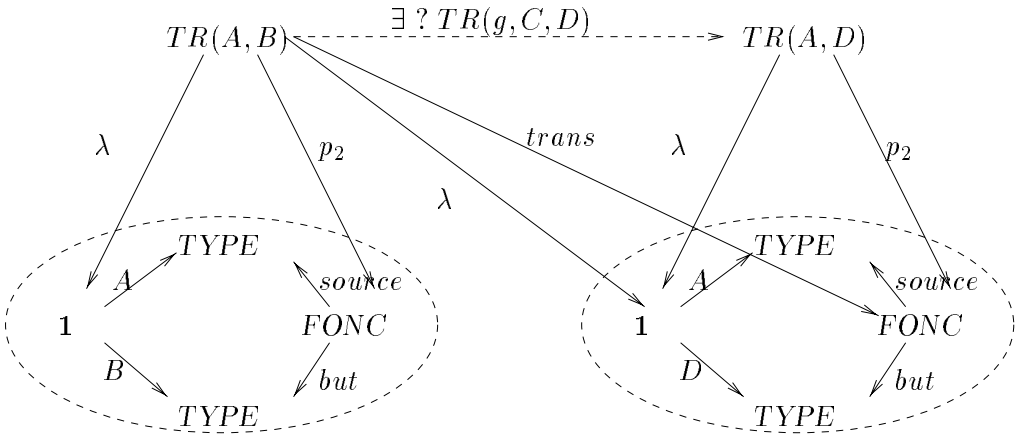


Dans un modèle ensembliste, l'objet $TR(A, B)$ s'interpréterait comme l'ensemble des fonctions de source A et de but B, ou plus simplement $Hom(A, B)$.

Dès que l'on dispose de deux fonctions Aldor $f : A \to B$ et $g : C \to D$, où le type C satisfait le type B, on dispose également dans $\mathbf{C}_{Aldor}$ des deux

objets $TR(A,B)$ et $TR(A,D)$. Comme $\mathbf{E}_{Aldor}$ est en particulier une esquisse des catégories, on va mettre en évidence dans $\mathbf{C}_{Aldor}$ l'existence d'une flèche $TR(g,C,D)$ de $TR(A,B)$ vers $TR(A,D)$ et qui traduit effectivement un changement de types sous l'action de g (en composant par la fonction g, on va passer du type B au type D).

La flèche $TR(g,C,D)$ existe dès que l'on a une flèche *trans* convenable :



On prouve l'existence de *trans* en utilisant les propriétés de factorisation et de cofactorisation des cônes limites dans $\mathbf{C}_{Aldor}$.

7.5.2 Notion d'héritage

On a un héritage direct entre deux catégories si ces deux catégories sont en relation de satisfaction en tant que types, au sens de la règle R_4. On étend alors la notion d'héritage à d'autres catégories en utilisant la transitivité de la relation de satisfaction avec les autres cas de satisfaction entre catégories.

L'hèritage entre catégories peut ainsi se traduire uniquement en termes de satisfaction de types dans $\mathbf{C}_{Aldor}$.

7.5.3 Notion d'appartenance

On parle d'appartenance d'un domaine à une catégorie lorsque le type du domaine, s'il s'agit d'une catégorie, satisfait la catégorie en question. On utilise en Aldor le prédicat `has`.

Là encore, cette notion peut donc être décrite en termes de satisfaction de types dans $\mathbf{C}_{Aldor}$.

7.5.4 Les conditionnelles

Une conditionnelle dans un langage informatique est en général une expression de la forme :

`if` (*condition*) `then` (*action*) `else` (*action*)

Il est possible en aldor d'écrire une conditionnelle comme pour n'importe quel langage : "if $n \neq 0$ then $\frac{1}{n}$ else 0" est une expression utilisable dans un programme. Mais aldor offre un autre mode d'utilisation des conditionnelles dans les expressions de catégories :

"`if ... has ... then ... else ...` "

Il y a trois contextes dans lesquels on peut inscrire cette conditionnelle :

- if *un domaine* has *une catégorie* then ... else ...
- if *un domaine* has *un attribut* then ... else ..., avec la notion d'attribut valable uniquement pour le langage Axiom,
- if *un domaine* has *une opération* then ... else ..., qui est une utilisation envisageable.

On peut ainsi conditionner la partie exportée d'un domaine en soumettant l'ajout (d'une opération, d'un attribut ou) d'une appartenance à une catégorie à la validation d'un test.

Exemple : Une catégorie des corps quotients.

```
QuotientFieldCategory(S:IntegralDomain) :  Category == Join(Field,
Algebra S, RetractableTo S, FullyEvalableOver S, DifferentialExtension S,
FullyLinearlyExplicitRingOver S, Patternable S,
FullyPatternMatchable S) with {
    -/              : (S,S) → % ;
    numer           : % → S ;
```

```
    denom           : % → S ;
    numerator       : % → % ;
    denominator     : % → % ;
    if S has StepThrough then StepThrough ;
    if S has RetractableTo Integer then
                    {RetractableTo Integer ;
                    RetractableTo Fraction Integer ; }
    if S has EuclideanDomain then
    {wholepart      : % → S}
    ...
}
```

Remarque : Lorsque l'on teste une propriété d'un domaine dans une conditionnelle, on teste en fait la partie exportée par ce domaine, et le relation nouvellement introduite entre les deux catégories n'est plus un héritage de construction (par un enrichisssement de type "`with`"), mais de conception.

Ici aussi le cadre des esquisses que l'on a choisi se révèle être particulièrement bien adapté puisque une conditionnelle s'exprime à l'aide d'une cofactorisation dans la théorie des esquisses. C'est cette propriété que l'on utilisera dans $\mathbf{C}_{Aldor}$, sur des types, avec le prédicat d'appartenance dont on vient de voir que la définition pouvait se faire en termes de satisfaction de types.

7.6 Conclusion

Le point crucial de ce travail consiste à remarquer que, pour avoir voulu élever le niveau de description des types, Aldor a introduit ce faisant une ambiguité, qui ne peut pas être levée par des considérations de typage.

Si la plupart des langages informatiques typés arrivent à décrire une expression par son type, il y a en revanche peu ou pas de satisfaction de types. En Aldor on a à la fois une satisfaction de types non triviale et des types qui sont valeurs de premier ordre. La contrepartie de cette complexité est paradoxalement une perte de richesse expressive des types et une vérification de validité des expressions qui se décompose donc en deux vérifications distinctes :

```
x : T == E
```

1. Le type de `x` doit satisfaire `T`.

2. Le type `T` doit être compatible avec la nature de `E`.

L'introduction de la notion de nature dans le système de typage d'Aldor permet certes de répondre au besoin de vérification de validité d'expressions, mais elle a également l'avantage de préciser le procédé d'évaluation d'un programme Aldor.

Un programme Aldor, dans notre formalisme, est une flèche dans $\mathbf{C}_{Aldor}$ de domaine $\mathbf{1}$. C'est une composée de flèches élémentaires et tout le problème de l'évaluation consiste à la réduire en une flèche "équivalente" dans $\mathbf{C}_{Aldor}$. On utilise entre autres des mécanismes de factorisation et cofactorisation. Le point problématique en Aldor est celui d'une flèche arrivant dans *EXPR* par application d'une fonction à un item (*itemcomp*). On ne sait pas a priori décider, en termes de typage, si l'on arrive dans *EXPR* par les objets de base, les fonctions, les domaines ou les catégories. C'st la donnée de la nature, déterminée par le corps de la fonction, qui résout le problème d'indécision. Ceci est obtenu par le fait qu'il n'y a qu'un nombre fini de constructeurs en Aldor pour les expressions sous forme primitive, et que chacun correspond à une nature déterminée.

L'esquisse $\mathbf{E}_{Aldor}$ est donc quasi-projective.

Bibliographie

[1] P. Ageron. *Sémantique catégorique des types : comprendre le système F*, Diagrammes 19 (1988)

[2] M. Barr, C. Wells. *Categories theory for computing Science*, Prentice Hall (1990)

[3] D. Bert, R. Echahed, P. Jacquet, M.L. Potet, J.C. Reynaud. *Spécification, Généricité, Prototypage : Aspects du langage LPG* , Technique et Science Informatiques, TSI (1995) 9(14) 1097-1129

[4] L. Coppey, C. Lair. *Leçons de théorie des esquisses (I)*, Diagrammes 12 (1984)

[5] L. Coppey, C. Lair. *Leçons de théorie des esquisses (II)* , Diagrammes 19 (1988)

[6] D. Duval, J.C. Reynaud. *Sketches and Computation I : basic definitions and static evaluation* , Math. Struct. in Comp. Science, Cambridge University Press (1994) vol 4 185-238

[7] D. Duval, J.C. Reynaud. *Sketches and Computation II : dynamic evaluation and applications* , Math. Struct. in Comp. Science, Cambridge University Press (1994) vol 4 239-271

[8] C. Ehresmann. *Esquisses et types de structures algébriques* , Bulletin de l'Institut Polytechnique, Iasi 14 (1968)

[9] J.A. Goguen, J.W. Thatcher, E.G. Wagner. *An initial algebra approach to the specification, correctness, and implementation of abstract data types* , Current Trends in Programming Methodology, Vol. IV : Data Structuring, R.T. Yeh ed., Prentice Hall (1978) 80-149

[10] R. Guitart, C. Lair. *Calcul syntaxique des modèles et calcul des formules internes* , Diagrammes 4 (1980)

[11] R. Guitart, C. Lair. *Limites et colimites pour représenter les formules* , Diagrammes 7 (1982)

[12] Hudak, Jones, Wadler. *Report on the programming language Haskell : a non-strict, purely functional language* , ACM SIGPLAN Notices (1992) 27(5)

[13] R. Jenks, R. Sutor. *AXIOM The Scientific Computation System*, Springer (1992)

[14] J.L. Krivine. *Lambda-calcul : types et modèles*, ERI Masson (1990)

[15] R. Lalement. *Logique, réduction, résolution*, ERI Masson (1990)

[16] S. Mac Lane. *Categories for the working mathematician*, Springer (1971)

[17] S.K. Lellahi. *Categorical abstract data types* , Diagrammes 21 (1989)

[18] Milner. *A proposal for Standard ML* , Proc. of the Symposium on Lisp anf Functional Programming, ACM (1984) 184-197

[19] S. Watt,R. Sutor, P. Broadbery, S. Dooley, P. Iglio, S. Morrison, J. Steinbach. *AXIOM Library Compiler - User Guide*, IBM Thomas J.Watson Research Center, The Numerical Algorithms Group Limited