Modélisation de problèmes
Approche fonctionnelle

Philippe Chatalic, Claude Delobel, Delia Kesner, Christine Paulin, Marie-Christine Rousset

6 juin 2003

Le but du cours est d'enseigner des méthodes de base pour la modélisation fonctionnelle de problèmes informatiques. Le cours insiste sur la structuration des données et la spécification d'opérations agissant sur ces données. On introduit en particulier les notions d'algorithmes, de types de données structurés, de récursivité et de complexité.

Un langage de description de haut niveau est introduit pour la modélisation du problème. Le cours s'appuie sur le langage de programmation fonctionnel CAML.

Transparent

Cours n^o 1
Problèmes et Algorithmes

Ce chapitre rappelle la distinction entre problème, algorithme et processus. On montre qu'il existe des problèmes (dits *indécidables*) qui ne peuvent être résolus par un traitement informatique. On insiste sur l'importance de structurer les données et les traitements pour décomposer les problèmes et ainsi faciliter leur résolution informatique. Ce point est illustré sur un exemple de problème concret: la gestion d'une bibliothèque. On introduit un nouveau paradigme de programmation fondé sur une *approche fonctionnelle de la conception et du codage d'algorithmes*, en le distinguant du paradigme de programmation impérative introduit en S2.

Fin 1.0

1.1 Problèmes, Algorithmes et processus

Transparent

Étant donné un problème à résoudre, on peut procéder en deux étapes :

1. Construire un *modèle abstrait* du problème et essayer de donner une *caractérisation de la façon dont on va le résoudre* pour obtenir le résultat escompté (i. e. le mode d'emploi)
2. *Résoudre effectivement* le problème conformément à la description précédente.

Fin 1.1

1.1.1 Algorithme

L'énoncé résultant de la première étape constitue ce que l'on appelle un *algorithme*. Il s'agit d'une description *finie* d'un enchaînement d'opérations ou de tâches *élémentaires* (i.e. que l'on sait déjà résoudre.

Origine Le mot algorithme vient du nom du mathématicien arabe Al-Khowarizmi [Bagdad VIIIème siècle]. À cette époque l'informatique n'existait pas.

1.1.2 Processus

La seconde étape consiste à effectuer les opérations mentionnées dans l'algorithme en respectant les règles d'enchaînement spécifiées par celui-ci. Cela correspond à l'exécution de l'algorithme.

On parle de *processus* engendré par l'algorithme.

Transparent

1.1.3 Différences entre algorithmes et processus

- Un même algorithme peut engendrer plusieurs processus différents.
- L'algorithme est une description finie d'un enchaînement d'opérations, le processus engendré par un algorithme peut correspondre à un enchaînement infini d'opérations.
  Exemple Énumérer tous les entiers positifs.

– Un algorithme est une description abstraite qui fait référence à des paramètres abstraits (en les nommant)

$$Moyenne = (Note\_Partiel + Note\_Exam)/2$$

Un processus associe à de tels paramètres des valeurs concrètes.

Fin 1.2

1.1.4 Exemples d'algorithmes

Transparent

Déterminer si un nombre est premier Soit n un nombre, tester pour chaque valeur de p comprise entre 2 et $n-1$ si p divise n. Si ce n'est le cas pour aucune valeur alors n est premier, sinon n n'est pas premier.

Amélioration 1: tester pour 2 et tous les nombres impairs entre 3 et $n-1$.
Amélioration 2: tester pour 2 et tous les nombres impairs entre 3 et $\sqrt{n}$.

Fin 1.3

Calculer la racine carrée d'un nombre Cet exemple sera développé ultérieurement.

1.1.5 Autres algorithmes

Les algorithmes ne sont pas forcément exécutés par des ordinateurs :
– recette de cuisine,
– mode d'emploi d'un appareil ménager,
– partition de musique,
– méthode utilisée par un technicien pour régler un téléviseur ou pour identifier une panne,
– mise à jour de l'heure d'un réveil matin,
– description d'un itinéraire.

1.2 Problèmes sans algorithme

Transparent

Question: *Étant donné un problème, peut-on trouver un algorithme pour le résoudre ?*

On a longtemps pensé pouvoir répondre à cette question par l'affirmative et que la principale difficulté était de trouver le formalisme adéquat. En fait Church et Turing ont montré dans les années 30 que ceci n'était pas vrai et qu'il existe des problèmes qu'aucun algorithme ne permet de résoudre: on dit de ces problèmes qu'ils sont *indécidables*.

Fin 1.4

Transparent

1.2.1 Justification de l'existence de problèmes sans algorithme

On considère les ensembles suivants:
– $\mathcal{F}$ est l'ensemble des fonctions (totales) sur les entiers.
– $\mathcal{A}$ est l'ensemble des algorithmes qui à partir d'un entier retournent un entier.

Un algorithme de $\mathcal{A}$ peut être vu comme un moyen de calculer une fonction de $\mathcal{F}$.

Un même algorithme ne peut pas calculer deux fonctions différentes.

On a donc une application de $\mathcal{A}$ dans $\mathcal{F}$. On va montrer qu'il existe des fonctions de $\mathcal{F}$ qui ne peuvent être codées par aucun algorithme, donc qui ne sont pas dans l'image de $\mathcal{A}$. Pour cela on montre que $\mathcal{A}$ est dénombrable et que $\mathcal{F}$ ne l'est pas.

Fin 1.5

Rappel Un ensemble est dit dénombrable s'il peut être mis en bijection avec $\mathbb{N}$ i.e. s'il a le même nombre d'éléments que $\mathbb{N}$.

Preuve que $\mathcal{A}$ est dénombrable

Un algorithme est décrit par un texte formé d'une suite finie de caractères. On peut construire une numérotation des algorithmes en considérant tous les textes formés d'un seul caractère, puis de deux caractères, ...et en ordonnant par exemple de manière lexicographique les suites de même longueur.

Preuve que $\mathcal{F}$ n'est pas dénombrable

On montre que supposer $\mathcal{F}$ dénombrable conduit à une contradiction :
- si $\mathcal{F}$ est dénombrable alors on peut numéroter ses éléments sous la forme $f_1, f_2, \ldots$
- soit g la fonction définie par $g(n) = f_n(n) + 1$,
- $g \in \mathcal{F}$ donc il existe k tel que $g = f_k$
- on a alors $g(k) = f_k(k) + 1 = f_k(k)$ d'où une contradiction

1.2.2 Applications

Il existe des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ qu'aucun algorithme ne permet de calculer. Plus généralement il existe des problèmes qu'aucun algorithme ne permet de résoudre.

Transparent

Un problème important en informatique concerne la caractérisation de ce qui est calculable (décidable) ou pas.

Exemple Il n'existe pas d'algorithme général qui permet de déterminer si une équation polynomiale à plusieurs variables et coefficients entiers naturels admet des solutions entières. L'équation $x^2 + y^2 = z^2$ admet pour solution entière $x = 3, y = 4$ $z = 5$, l'équation $x^2 = 2$ n'admet pas de solution entière mais on ne peut pas décider en général en fonction des coefficients si une solution existe.

Fin 1.6

1.3 Décomposition de problèmes et structuration des données

Transparent

La conception d'un algorithme nécessite :
- d'identifier les différents niveaux d'abstraction du problème
- pour chaque niveau d'abstraction, d'identifier les différentes tâches caractéristiques de ce niveau
- pour chaque tâche, de la décrire sous la forme d'un enchaînement de tâches de niveau d'abstraction inférieur.

Fin 1.7

L'approche par décomposition est essentielle pour pouvoir maîtriser la résolution de problèmes complexes. Les programmes informatiques peuvent comporter plusieurs centaines de milliers de lignes : la décomposition qui peut sembler superflue sur des programmes d'une centaine de lignes, devient un facteur essentiel pour la correction du code développé, son efficacité et sa réutilisation.

Intuitivement chaque fois que l'on décompose un problème, on espère pouvoir se ramener à des problèmes plus simples, plus faciles à traiter et à modéliser et que l'on peut aborder séparément.

Cette *Modélisation par raffinages successifs* est appelée *approche descendante*.

Transparent

1.3.1 Modularité

L'identification des différents niveaux d'abstractions permet parfois d'identifier certaines sous-tâches utilisées dans la description de plusieurs tâches à des niveaux plus élevés.

Exemple : plusieurs algorithmes font appel au calcul du minimum de deux nombres. On n'inclut pas la méthode de calcul du minimum dans la description de ces algorithmes car ils doivent pouvoir fonctionner avec plusieurs méthodes de calcul de minimum, et la même méthode de calcul du minimum peut servir à plusieurs algorithmes.

Fin 1.8

On évite de réécrire plusieurs opérations identiques, on a donc une *approche plus modulaire*.

La réutilisation de tâches déjà spécifiées pour décrire des tâches plus complexes, s'appelle *l'approche ascendante*.

Transparent

1.3.2 Niveaux d'abstraction de données

La construction d'un modèle abstrait du problème conduit également à préciser la structure des données traitées par les différentes tâches.

Là-encore une bonne modélisation passe généralement par l'identification de différents niveaux d'abstraction dans l'organisation des données. Une donnée complexe peut alors se décrire en termes de composants qui relèvent d'un niveau d'abstraction inférieur.

Fin 1.9

Par exemple, un algorithme manipulant des polynômes à coefficients rationnels, s'appuiera sur une représentation des rationnels qui elle-même pourra s'appuyer sur une représentation d'entiers arbitrairement grands.

Transparent

1.3.3 Liens entre données et traitements

Lorsque l'on cherche à modéliser un problème on constate que l'organisation des données n'est généralement pas déconnectée de celle des tâches ou traitements que l'on a identifiés.

Chaque niveau d'abstraction correspond à un certain niveau de détail
- dans l'organisation des données,
- dans l'organisation des traitements (i.e. des tâches)

Fin 1.10

1.4 Exemple de la gestion d'une bibliothèque

Nous illustrons les notions introduites précédemment sur l'exemple de la modélisation de la gestion d'une bibliothèque.

1.4.1 Objectifs

Transparent

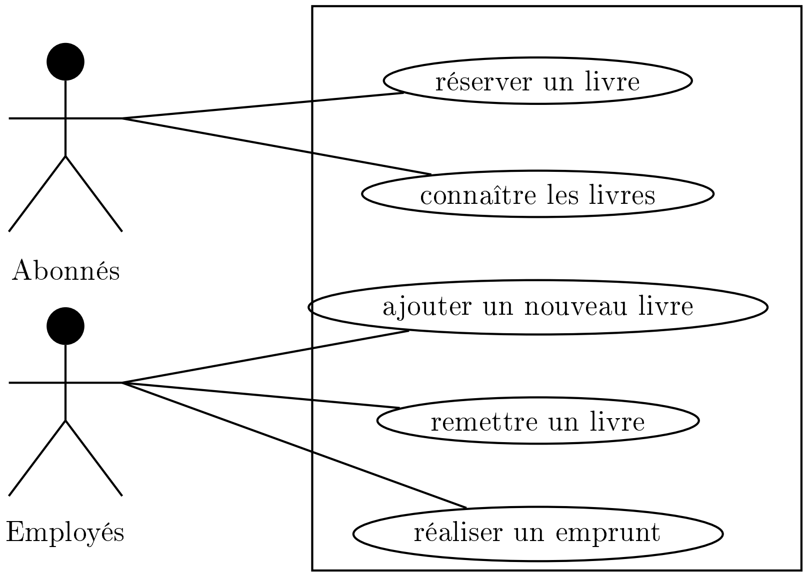
Gestion d'une bibliothèque : cahier des charges des tâches que l'on cherche à modéliser.
- Permettre aux abonnés de connaître le fond documentaire
- Permettre aux abonnés de réserver un livre
- Permettre aux abonnés de connaître les livres empruntés
- Les prêts sont effectués par les personnels de bibliothèque
- Les employés de la bibliothèque ajoutent de nouveaux livres et remettent les livres rendus.

Fin 1.11

1.4.2 Les tâches du système

Le diagramme suivant illustre l'organisation de la gestion de la bibliothèque.

Transparent



Fin 1.12

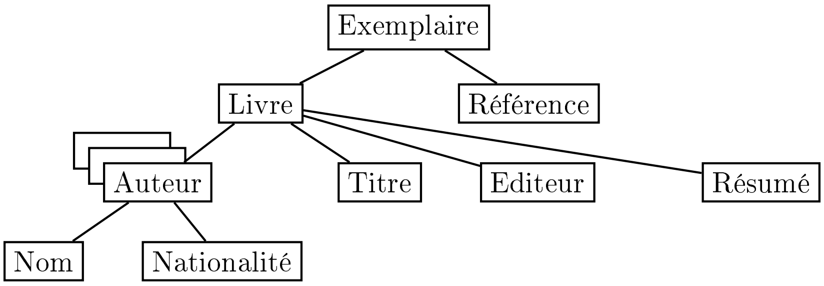
1.4.3 Modélisation des données

Avant de préciser les tâches à effectuer, il est nécessaire de modéliser les données de la bibliothèque. On est amené à représenter différentes entités.

Livre et exemplaire

On modélise les notions de livre et d'exemplaire sachant qu'il peut y avoir plusieurs exemplaires du même livre.

Transparent



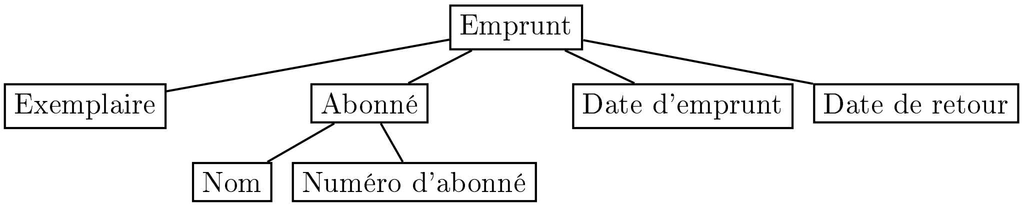
Pour une tâche donnée, tout ou partie de ce diagramme est utile, par exemple pour connaître les livres, on peut se limiter à *Livre*, *Auteur* et *Titre*.

Fin 1.13

Emprunt

Transparent

La modélisation d'un *emprunt* met en jeu l'exemplaire, l'abonné et les dates d'emprunt et de retour.

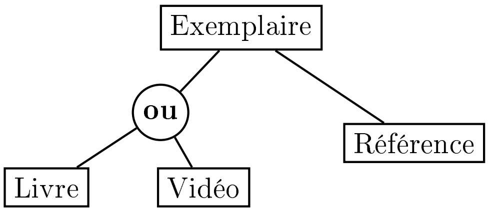


Fin 1.14

1.4.4 Extension du concept d'exemplaire

Transparent

On peut être amené à modifier la structuration des données, par exemple, un exemplaire pourra être soit un livre soit une vidéo.



Fin 1.15

1.4.5 Décomposition des tâches

Transparent

La réservation d'un livre est une tâche qui peut se décomposer.

Réservation

L'abonné se présente devant un terminal
L'abonné choisit une opération (réservation)
L'abonné donne son nom
Le système identifie l'abonné
L'abonné choisit un livre
Le système effectue l'emprunt

Fin 1.16

Transparent

La tâche *choisir un livre* peut elle-même se décomposer.

Choisir un livre

Sous quelle forme ?	
À partir du titre	À partir du nom de l'auteur
L'abonné confirme son choix	
Exemplaire disponible ?	

Fin 1.17

1.4.6 Cas exceptionnels

Le déroulement d'une tâche peut varier suivant les situations. Par exemple, après la tâche d'identification d'un abonné, le système peut rejeter l'abonné. Autre exemple, il n'y a plus d'exemplaire de disponible, dans ce cas l'abonné doit pouvoir recommencer la tâche de choisir un livre.

Il est toujours bon de regarder les différentes exécutions possibles pour une tâche.

1.5 Différents paradigmes de programmation

Pour modéliser complètement la résolution informatique d'un problème, il faut indiquer sous la forme d'un algorithme la façon dont les différentes tâches doivent s'effectuer. Il faut ensuite coder cet algorithme dans un langage de programmation pour le rendre exécutable sur machine.

1.5.1 Langage de description

Un algorithme peut être codé dans n'importe quel langage de programmation mais, selon le langage de programmation choisi, le codage sera plus ou moins direct.

Transparent

Pour pouvoir décrire un algorithme, il faut utiliser un *langage de description* permettant d'exprimer :
- les objets et tâches primitives
- l'enchaînement des tâches pour réaliser des tâches complexes (séquence, conditionnelle, itération, ...)
- les situations exceptionnelles
- des mécanismes d'abstraction qui permettent de référencer un objet ou une tâche complexe par un nom sans tenir compte des détails.

Fin 1.18

1.5.2 Programmation fonctionnelle et impérative

Pour la description et le codage d'algorithmes, on distingue differentes approches qu'on qualifie de *paradigmes de programmation*.

Transparent

Le paradigme de programmation impérative correspond à une approche où les algorithmes et les programmes sont vus comme des actions modifiant le contenu de la mémoire de la machine. L'action de base est l'affectation de variables où une variable représente à un certain niveau d'abstraction le contenu d'une case mémoire, et l'affectation à une opération qui modifie ce contenu.

Le paradigme de programmation fonctionnelle est une approche différente dans laquelle les algorithmes et les programmes sont des fonctions qu'on évalue sur des données à l'aide de la machine vue comme un moyen de calcul. Les langages fonctionnels (de description d'algorithme ou de programmation) reposent sur des primitives permettant de décrire des expressions complexes modélisant des fonctions. L'exécution d'un algorithme ou d'un programme consiste alors à évaluer les expressions décrivant les fonctions sur des données fournies comme paramètres d'appel des fonctions pour produire des résultats.

Fin 1.19

Pour modéliser la résolution informatique d'un problème, on peut adopter indifféremment une approche impérative ou fonctionnelle. Il existe d'ailleurs d'autres paradigmes de programmation possibles (paradigme de programmation "objet" à la Java, ou paradigme de programmation logique). Tous ces paradigmes de programmation sont équivalents : aucun ne permet de résoudre un problème qu'un autre ne pourrait pas résoudre.

Cependant, chaque paradigme a ses avantages. Ainsi, le paradigme de programmation impérative est plus proche du langage machine. Par contre, il est souvent plus facile de prouver qu'un algorithme est correct quand il est décrit selon une approche fonctionnelle plutôt que selon une approche impérative. Un algorithme est correct quand on peut garantir à partir de sa description que son exécution correspond bien à la tâche annoncée dans sa spécification.

1.5.3 Exemple

Transparent

Tâche visée: étant donné un entier n calculer $n! = n \times (n-1) \times \cdots \times 1$

Approche impérative :

```
action fact(n: entier ; résultat r: entier)
{r = n!}
  r <- 1
  tant que n >= 2 faire
    début
      r <- n*r
      n <- n-1
    fin
```

Fin 1.20

Transparent

Approche fonctionnelle :

```
fonction fact(n:entier) : entier
{fact(n) = n!}

fact(n) ->
  si n = 0 alors:  1
           sinon: n*fact(n-1)
```

Fin 1.21

Dans le cas impératif, il faut montrer que, si r est une variable, l'exécution de `fact`(n, r) se terminera avec dans la variable r la valeur $n!$. Pour cela, il faut établir un invariant de boucle (propriété des variables préservée lors de l'exécution du corps de la boucle). Dans le cas de `fact` un invariant possible est $r \times n! = n_0!$ où r et n désignent les valeurs des variables r et n au point de programme considéré et où n_0 représente la valeur de n au moment de l'appel de l'action `fact`. Le fait de devoir raisonner sur les valeurs successives des variables complique les preuves de propriétés des programmes impératifs.

Dans le cadre fonctionnel, l'exécution suit l'évaluation des expressions comme en mathématiques. Les propriétés $0! = 1$ et $n! = n \times (n-1)!$ suffisent à établir la correction de la description fonctionnelle de la `fact`.

Transparent

1.5.4 Différences entre paradigme fonctionnel et impératif

Dans l'approche impérative, une variable représente une adresse en mémoire dont la valeur peut être modifiée au cours du calcul. L'action principale d'un programme est de modifier les valeurs des variables. Pour raisonner sur de tels programmes, il faut faire référence à la valeur des variables à un certain moment de l'exécution.

L'approche fonctionnelle est plus proche de la pratique mathématique, une variable est simplement un nom pour une valeur qui ne peut être modifiée par le calcul (sauf si la variable est redéfinie). L'action d'un programme est de construire de nouvelles valeurs.

L'intérêt de l'approche fonctionnelle est que la valeur représentée par une expression se calcule simplement en fonction des valeurs représentées par ses sous-expressions. Ceci permet de prouver que l'algorithme ainsi décrit réalise bien le calcul de la fonction spécifiée.

Fin 1.22

Transparent

L'exécution itérative en boucle des langages impératifs est réalisée à l'aide de fonctions récursives dans le paradigme fonctionnel. La programmation à l'aide de fonctions récursives seront étudiées en détail dans le chapitre 3.

Il est bien sûr possible d'écrire des fonctions récursives dans des langages de programmation impératifs comme Pascal et des variables modifiables ou des boucles dans des langages de programmation fonctionnels tels que Ocaml. Cependant, dans le cadre de ce cours, nous utiliserons un paradigme fonctionnel pur.

Fin 1.23

Transparent

Cours n^o 2
Modélisation fonctionnelle de problèmes

Ce cours introduit les langages fonctionnels de spécification, de réalisation et de codage considérés dans ce module. Il est centré sur la modélisation des données de base et des opérations associées, ainsi que sur la modélisation d'expressions conditionnelles, de composition de fonctions (mécanismes d'abstraction pour le nommage d'expressions).

Fin 2.0

2.1 Principes de base pour la modélisation fonctionnelle d'un problème

Transparent

2.1.1 Cycle de vie d'une modélisation

La modélisation d'un problème se fait dans notre approche à l'aide de fonctions. On distinguera quatre étapes.

- Spécification : introduit des ensembles de données et des fonctions sur ces données, on décrit dans le langage courant ce que représentent ces objets.
- Réalisation : explicite la structure des données et la manière de calculer le résultat des fonctions en utilisant un langage de description algorithmique.
- Codage : implante les données et les fonctions sur une machine en utilisant un langage de programmation.
- Validation : les fonctions codées sont testées sur des exemples apropriés pour vérifier l'adéquation avec la spécification.

Fin 2.1

2.1.2 Intérêt de distinguer spécification, réalisation et codage

- Un même problème peut être modélisé de différentes manières.
- Une même réalisation peut être codée :
  - de différentes manières (efficacité)
  - dans différents langages de programmation
- Le langage de réalisation peut être plus souple que les langages de programmation (ambiguïté) et étendu si nécessaire.

On est amené à distinguer différents types d'erreurs :

- au niveau de la spécification : erreur de modélisation;
- au niveau de la réalisation : erreur de conception;
- au niveau du codage : erreur de programmation.

On doit pouvoir se convaincre de la correction de la démarche à chaque étape.

2.2 Spécification

Domaines Une spécification peut introduire des ensembles de données qui sont appelés des *domaines.*

Les machines fournissent des domaines primitifs tels que les entiers, les flottants ou les chaînes de caractères. Les programmes peuvent manipuler des domaines plus complexes pour représenter par exemple une date, un vecteur ou une matrice.

Constantes Les constantes sont des noms associés à certaines valeurs d'un domaine. Un domaine primitif comme celui des entiers est spécifié par un ensemble de constantes. Nous verrons dans la suite que les valeurs d'un domaine complexe structuré sont plutôt décrites à l'aide de *constructeurs* qui sont des fonctions permettant de construire les valeurs du domaine complexe à partir de valeurs plus simples.

Fonctions Les machines exécutent des transformations sur les données. Dans l'approche fonctionnelle, ces transformations sont vues comme des fonctions qui prennent en argument des *paramètres* appelés aussi *entrées* et renvoient un *résultat*. Dans la phase de spécification on doit décrire pour chaque fonction :

- sa *signature*. Celle-ci est composée du *nom* de la fonction, des noms et domaines des paramètres ainsi que du domaine du résultat de la fonction. Le *profil* d'une fonction décrit uniquement les domaines des paramètres et du résultat de la fonction.
- son *sens*. Il définit par une phrase en français ce que la fonction est supposée calculer (par exemple par une relation entre les données en entrée et les données en sortie de la fonction). Il est particulièrement important de décrire les cas exceptionnels.
- des *propriétés* et *tests* : dès la phase de spécification, il est utile de préciser les propriétés vérifiées par les fonctions et de définir des jeux de tests correspondant à des entrées significatives (cas exceptionnels mais aussi cas courant) en donnant le résultat attendu.

Le choix des domaines, constantes et fonctions se fait en fonction du problème à traiter.

2.2.1 Notations

Le modèle général d'une spécification est :

Transparent

SPÉCIFICATION:	
{Domaines}	
dom	{indiquer les données représentées par le domaine *dom*}
...	
{Constantes}	
a : *dom*	{constante a du domaine *dom*, indiquer ce qu'elle représente}
...	
{Fonctions}	
f$(x_1 : t_1; \ldots ; x_n : t_n) \to t$	{Fonction f de profil $(t_1; \ldots ; t_n) \to t$}
...	{Indiquer ce que fait cette fonction}
{Propriétés}	
f$(a_1, \ldots , a_n) = b$	{Jeu de tests}

Fin 2.2

Une spécification peut contenir 0 ou plusieurs déclarations de domaines, de constantes ou de fonctions. Lorsqu'il y a plusieurs constantes $a_1, \ldots , a_p$ dans le même domaine *dom* on écrira plus simplement $a_1, \ldots , a_p$: *dom*. De même si plusieurs paramètres d'une fonction f on le même domaine on pourra écrire $x_1, \ldots , x_k : t$ plutôt que $x_1 : t; \ldots ; x_k : t$.

2.2.2 Exemples

Transparent

Exemple 1 : date On spécifie un domaine `date` représentant les dates à partir du 1er janvier 1900.

On spécifie des constantes représentant des dates comme `jour_de_l'an_1900` ou bien `st-sylvestre_1999` représentant respectivement le 1/01/1900 et le 31/12/1999.

On spécifie une fonction `absolue` qui prend en entrée une `date` et renvoie un entier représentant le nombre de jours écoulés depuis le 1er janvier 1900 et une fonction `plus_ancien` qui étant données deux *dates* d_1 et d_2 renvoie vrai si la date d_1 est strictement antérieure à la date d_2.

Fin 2.3

Transparent

Notation

SPÉCIFICATION:
{Domaines}
{`date` représente le domaine des dates à partir du 1er janvier 1900}
`date`
{Constantes}
`jour_de_l'an_1900`,`st-sylvestre_1999`:`date`
{Fonctions}
`absolue`(d:`date`) →`entier`
{associe à la date d le nombre de jours écoulés depuis le 1er janvier 1900}
`plus_ancien`(d1,d2:`date`) →`booléen`
{renvoie `vrai` si $d1$ est une date plus ancienne que $d2$ }

Fin 2.4

Transparent

On pourra effectuer les tests suivants :

SPÉCIFICATION:	
{Propriétés}	
`absolue`(`jour_de_l'an_1900`)	$= 0$
`absolue`(`st-sylvestre_1999`)	$= 100 \times 365 + 24 = 3674$
`plus_ancien`(`jour_de_l'an_1900`,`st-sylvestre_1999`)	= `vrai`
`plus_ancien`(d,d)	= `faux`

Fin 2.5

Transparent

Exemple 2 : vecteurs On spécifie le domaine des `vecteurs` représentant des vecteurs à coordonnées entières. On peut spécifier les fonctions suivantes :
- `id_vec` prend en argument un entier n et une valeur a et renvoie le vecteur à n coordonnées toutes égales à la valeur a
- `produit_scalaire` prend en argument deux vecteurs et renvoie un entier correspondant à leur produit scalaire;
- `taille` prend en argument un vecteur et renvoie un entier représentant sa taille (nombre de coordonnées).

Fin 2.6

Transparent

SPÉCIFICATION:
{Domaines}
{Le domaine `vecteur` représente des vecteurs à coordonnées entières}
`vecteur`
{Fonctions}
`id_vec`(n:`entier`;a:`entier`) →`vecteur`
{représente un vecteur de longueur n dont toutes les coordonnées valent a}
`taille`(v:`vecteur`) →`entier`
{calcule la taille du vecteur v}
`produit_scalaire`(v1,v2:`vecteur`) →`entier`
{calcule le produit scalaire de $v1$ et $v2$ si ce sont des vecteurs de même longueur, échoue sinon}

Fin 2.7

Transparent

On pourra vérifier les propriétés suivantes :

SPÉCIFICATION:
{Propriétés}

`produit_scalaire`(`id_vec`(2,1),`id_vec`(2,2))	= 4
`produit_scalaire`(`id_vec`(2,1),`id_vec`(3,2))	= `erreur`
	(tailles différentes)
`taille`(`id_vec`(n,x))	= n
`taille`(`id_vec`(2,1))	= 2

Fin 2.8

Transparent

Exemple 3: bibliothèque On spécifie des domaines `Exemplaire, Référence, Livre, Auteur, Titre, Editeur, Résumé` représentant les notions correspondantes.

Les fonctions de base sont :
- `livre_de_exemplaire` qui étant donné un exemplaire, renvoie le livre correspondant;
- `réference_de_exemplaire` qui étant donné un exemplaire, renvoie la référence;
- `titre_de_livre` qui étant donné un livre renvoie son titre;
- ...

Fin 2.9

Transparent

Exemple 4: Euro On souhaite modéliser des sommes d'argent exprimées en francs ou en euros. Afin d'éviter les confusions, il est recommandé d'introduire deux domaines `franc` et `euro` représentant respectivement des sommes d'argent en francs ou en euros. On spécifiera deux fonctions d'addition l'une s'effectuant sur le domaine `franc` et l'autre sur le domaine `euro`. On spécifiera également les fonctions `franc_vers_euro` et `euro_vers_franc` de conversion entre monnaies.

Fin 2.10

Transparent

Exemple : francs et euros - spécification

SPÉCIFICATION:	
{Domaines}	
`franc`	{représente une somme d'argent en francs}
`euro`	{représente une somme d'argent en euros}
{Constantes}	
`un_euro : euro`	{un euro}
`un_franc : franc`	{un franc}
`taux_euro : franc`	{le coût d'un euro en francs}

Fin 2.11

Transparent

SPÉCIFICATION:	
{Fonctions}	
`cons_f: (x:réel)` $\rightarrow$ `franc`	{la somme de x francs}
`cons_e: (x:réel)` $\rightarrow$ `euros`	{la somme de x euros}
`add_f: (x,y:franc)` $\rightarrow$ `franc`	{additionne deux sommes x et y en francs}
`add_e: (x,y:euro)` $\rightarrow$ `euro`	{additionne deux sommes x et y en euros}
`franc_vers_euro: (x:franc)` $\rightarrow$ `euro`	{convertit x francs en euros, arrondi au centime}
`euro_vers_franc: (x:euro)` $\rightarrow$ `franc`	{convertit x euros en francs, arrondi au centime}

Fin 2.12

Transparent

SPÉCIFICATION:
{Propriétés}
`euro_vers_franc(un_euro)=taux_euro`
`franc_vers_euro(taux_euro)=un_euro`
`add_e(un_euro,un_euro)=cons_e(2)`
`add_f(un_franc,un_franc)=cons_f(2)`
...

Fin 2.13

2.3 Réalisation

Domaines La réalisation décrit la représentation des données. Une donnée peut être représentée par des objets primitifs (entiers, chaînes de caractères) ou bien des objets complexes déjà définis ou encore par composition d'objets existants (couple, ensemble fini de valeurs,...)

Ainsi les constantes et les fonctions sur un domaine D dont la réalisation utilise un domaine D' pourront elles-mêmes être réalisées à l'aide des constantes et fonctions associées au domaine D'.

Transparent

Exemple : dates Une date pourra se représenter de nombreuses manières différentes :
- un entier : nombre de jours écoulés depuis une date initiale ou bien juxtaposition du jour, du mois et de l'année;
- une chaîne de caractères;
- un triplet formé de trois entiers représentant le jour, le mois et l'année;
- un couple formé de deux entiers représentant l'année et le numéro du jour dans l'année.
- ...

Fin 2.14

Dans le cas du franc et de l'euro, on choisira de représenter les deux domaines par des réels, les additions en francs ou en euros seront réalisées par la même fonction arithmétique.

Constantes La réalisation des constantes consiste à associer à chaque nom de constante du domaine D une expression représentant une valeur du domaine ayant servi à réaliser D.

Fonctions La réalisation décrit la méthode de calcul qui permettra de fournir le résultat de la fonction pour n'importe quelle donnée dans le domaine d'entrée de la fonction. Ces calculs sont décrits par une déclaration

$$f : (x_1, \dots, x_n) \mapsto e$$

dans laquelle f est le nom de la fonction, $x_1, \dots, x_n$ sont les n paramètres de la fonction (décrits dans la signature) et e est une expression mentionnant les x_i décrivant le calcul du résultat de la fonction. Cette notation est analogue à la pratique mathématique où une fonction f peut être décrite par l'équation :

$$f : (x, y) \mapsto x^2 + y^2$$

Ce calcul peut se faire par cas suivant les sous-domaines auxquels appartiennent les entrées de la fonction : dans notre exemple on pourrait distinguer le cas $x = 0$ ou bien $x < y$. Le calcul peut également combiner d'autres fonctions primitives ou définies par l'utilisateur.

2.3.1 Notations

Une réalisation est toujours associée à une spécification. Pour chacun des noms introduits dans la spécification elle indiquera si l'objet correspondant est abstrait ou bien elle lui associera une expression de notre langage de description. Le modèle général d'une réalisation est :

Transparent

RÉALISATION:	
{Domaines}	
dom = d	*domaine déjà connu*
...	
{Constantes}	
a = v	*valeur dans le domaine d*
...	
{Fonctions}	
$f : (x_1, \dots, x_n) \mapsto e$	
...	

Fin 2.15

2.3.2 Exemple : minimum

Soit à calculer le minimum de 4 nombres. Cette fonction notée `min4` prend en argument 4 entiers et renvoie un entier. On peut la réaliser en se ramenant au calcul du minimum de deux entiers : La Fonction `min` prend en argument deux entiers et renvoie un entier, elle peut se réaliser sans faire de tests, uniquement à l'aide des fonctions arithmétiques d'addition, de soustraction et valeur absolue (remarque : il est fortement déconseillé d'implémenter le min de cette façon).

Transparent

Specification :

`min`(a,b : entier) →entier

`min4`(a,b, c,d : entier) →entier

Realisation :

`min` : $(a, b) \mapsto (a + b - abs(a - b))/2$

`min4` : $(a, b, c, d) \mapsto \mathtt{min}(\mathtt{min}(a, b), \mathtt{min}(c, d))$

Fin 2.16

2.3.3 Exemple : francs et euros

L'exemple des francs et des euros peut se réaliser de la manière suivante :

Transparent

Réalisation:
`franc` = `réel`
`euro` = `réel`
`un_euro`= 1
`un_franc`= 1
`taux_euro`= $6,55957$
`cons_f`: $(x) \mapsto x$
`cons_e`: $(x) \mapsto x$
`add_f`: $(f_1, f_2) \mapsto f_1 + f_2$
`add_e`: $(e_1, e_2) \mapsto e_1 + e_2$
`franc_vers_euro:`$(f)\mapsto$`arrondi(`f`/taux_euro,2)`
`euro_vers_franc:`$(e)\mapsto$`arrondi(`e`*taux_euro,2)`

Fin 2.17

2.3.4 Domaines et fonctions abstraits

Lorsque l'on modélise un problème, on ne cherche pas forcément à réaliser tous les domaines et les fonctions. En effet on peut supposer que certains domaines et fonctions sur ces domaines seront fournis par une bibliothèque, ou bien écrits par un collègue. Il n'est pas souhaitable alors de faire des hypothèses sur la manière dont ces données seront représentées ou sur la manière dont les fonctions seront implantées. Si les fonctions primitives existent dans tous les langages de programmation, le codage de données structurées (par exemple ensembles finis, vecteurs, dates,...) et des fonctions de base manipulant ces objets dépend lui énormément du langage de programmation choisi. Plutôt que de paraphraser dans le langage de description les constructions de types structurés du langage de programmation, nous ne ferons que spécifier ces domaines et les fonctions s'y rapportant (cf chapitre 4).

Il est possible de vérifier la correction d'une réalisation même si elle dépend de certains domaines et fonctions abstraits. Par contre pour exécuter le programme, il faudra que toutes les fonctions soient implantées. Les domaines et fonctions laissés abstraits dans la phase de réalisation devront être codés dans le langage de programmation.

Notation Lorsque dans une réalisation, on choisit de laisser abstrait un domaine, une constante ou une fonction de la réalisation on pourra l'indiquer explicitement en écrivant :

$$dom = <abstrait> \quad f = <abstrait>$$

Exemple Dans le cas des dates, on peut choisir de laisser ce domaine abstrait, auquel cas on ne peut pas réaliser la fonction `absolue`. Par contre il est possible de réaliser le fonction `plus_ancien`. On écrira donc :

Transparent

RÉALISATION:
`date` = $<abstrait>$
`absolue` = $<abstrait>$
`plus_ancien` : $(d1, d2) \mapsto$`absolue`(d1) < `absolue`(d2)

Fin 2.18

2.3.5 Expressions licites

Les domaines représentent des ensembles de valeurs. Les expressions représentent des compositions de fonctions permettant de calculer des valeurs.

Certaines expressions ne sont pas licites, par exemple la fonction *cosinus* appliquée à une chaîne de caractères.

Si on se donne une expression e et un domaine d, on aimerait savoir si le calcul de l'expression e fournira une valeur du domaine d. C'est le critère de bonne formation des expressions. On dira que *e est une expression dans le domaine d* (ou plus simplement est dans le domaine d) si e satisfait les règles de bonne formation des expressions que nous définissons pour chaque nouvelle expression.

Bonne formation de l'application d'une fonction

La règle de bonne formation des expressions est la suivante : si le profil de f est $(d_1; \ldots; d_n) \rightarrow d$ et si e_1 est une expression dans le domaine d_1,..., e_n est une expression dans le domaine d_n alors $f(e_1, \ldots, e_n)$ est une expression bien formée dans le domaine d.

Cette règle est analogue à la règle d'application d'une fonction à son argument en mathématique : si f est une fonction de A dans B alors $f(x)$ n'est défini que pour $x \in A$ et dans ce cas $f(x) \in B$. Vérifier que les expressions écrites sont bien formées permet de détecter très tôt des erreurs dans l'écriture d'une réalisation.

2.4 Codage

Le codage utilise les primitives offertes par le langage de programmation pour coder en machine les structures de données et les algorithmes décrits dans l'étape de réalisation. Les domaines et fonctions abstraits devront soit correspondre à des bibliothèques de programmes du langage, soit être codés directement.

2.4.1 Le langage de programmation CAML

Le codage se fera dans le langage de programmation CAML.

CAML est un langage fonctionnel fortement typé, chaque expression bien formée a un type unique qui est calculé automatiquement. Ce type correspond aux domaines introduits dans notre langage de description. De même, lors de la déclaration d'une fonction, les domaines des arguments et du résultat (ce que nous appellerons le *profil* de la fonction) est automatiquement calculé. Cette particularité fait que de nombreuses erreurs de programmation peuvent être détectées automatiquement et que le développement d'exemples significatifs peut être fait très rapidement (cf projet et option LGL de S4).

CAML est proche de notre langage de description. Certaines constructions de notre langage de description seront simplement paraphrasées dans CAML. Mais nous utiliserons également des constructions CAML (comme les types structurés et le filtrage) qui n'ont pas de correspondant dans notre langage de description. Réciproquement certaines constructions du langage de description (les sous-domaines où le `selon le cas`) n'ont pas de correspondant direct dans CAML.

Il existe une famille de dialecte ML (SML, Camllight, OCAML, Moscow ML). Nous utiliserons OCAML. Ce langage est performant et disponible sur de nombreuses plateformes (cf `http://caml.inria.fr`).

2.4.2 Domaines, profils et types

Dans le codage en CAML, les domaines des objets et les profils des fonctions vont correspondre à des types. Les constantes correspondront à des déclarations d'identificateur et les fonctions à des déclarations de fonction.

Le modèle général est le suivant :

Transparent

```
(* Constantes *)
(* a représente ... *)
   let a = v;;
(* Fonctions *)
(* f(x_1 : t1,..,x_n :t_n) -> t  représente ...*)
   let f x_1 ... x_n = e;;
(* Tests *)
(* résultat attendu f(a_1,..,a_n)=b *)
   (f a_1 ... a_n) = b;;
```

Fin 2.19

Transparent

Mise en garde Les notations de CAML diffèrent de celles du langage de description :

Langage de description	CAML
$f : (x_1, \ldots, x_n) \mapsto e$	`let` f x1 ... xn = e
$f(e_1, \ldots, e_n)$	(f e1 ... en)

Fin 2.20

Les déclarations de CAML servent à introduire des noms pour représenter une constante ou une fonction :

$$\texttt{let}\ id = e;;$$

introduit la constante id pour représenter la valeur de l'expression e. Le nom id représente cette valeur dans toutes les expressions qui suivent jusqu'à la prochaine déclaration du même nom id.

$$\texttt{let}\ f\, x_1 \ldots x_n \;=\; e;;$$

introduit la fonction f à n paramètres $x_1, \ldots, x_n$ dont la valeur est calculée à partir de l'expression e.

Un programme CAML est constitué d'un certain nombre de définitions de constantes et fonctions séparées pas un double point-virgule et suivis par une expression à évaluer.

Le résultat de l'évaluation du programme est la valeur de cette expression.

Un programme CAML peut et *doit* comporter des commentaires. Un commentaire est une phrase qui commence par `(*` et termine par `*)`.

Principe Faire apparaître dans un commentaire la spécification de la fonction avant le codage.

Controle de type en CAML Les domaines du langage de réalisation correspondent aux types du langage de programmation. CAML est capable de calculer un type pour chaque expression ou déclaration. Si une expression ne respecte pas les règles de bonne formation alors une erreur est déclenchée :

Transparent

```
# 1+2;;
- : int = 3

# 2+true;;
This expression has type bool but is here used with type int

# let plus1 x = x+1;;
val plus1 : int -> int = <fun>

# let plus x y = x+y;;
val plus : int -> int -> int = <fun>

# (plus 2 3);;
- : int = 5
# (plus 2 true);;
This expression has type bool but is here used with type int
```

Fin 2.21

2.4.3 Validation

Il s'agit d'exécuter les fonctions qui ont été codées sur des exemples choisis au moment de la spécification. Ces tests ne garantissent pas la correction totale du programme (sauf à tester toutes les entrées ce qui n'est pas possible si elles sont en nombre infini) mais ils permettent de détecter des erreurs dans la réalisation ou le codage.

Il est important de calculer le résultat des tests *avant l'exécution* afin de ne pas se laisser influencer par le résultat de son programme. On peut calculer ces résultats attendus en utilisant un autre outil ou bien choisir des valeurs judicieuses permettant un calcul à la main. Enfin, il est essentiel de prévoir des tests pour tous les cas exceptionnels des entrées, en particulier ceux devant provoquer une erreur.

Pour faciliter les tests, il est commode d'introduire des constantes représentant des valeurs particulières du domaine.

Présentation de tests Pour présenter un jeu de test, on procèdera de la manière suivante :

- Donner des noms à des expressions qui interviendront dans les tests, ces expressions peuvent être choisies de manière aléatoire pour tester des cas généraux ou au contraire peuvent être des valeurs particulières permettant de tester des propriétés spécifiques du programme comme des cas limites, des cas d'échecs ou des cas simples.

– Indiquer dans un commentaire pour chaque test, le but du test (cas particulier, cas général) ainsi que le résultat attendu.
– Donner l'expression de programmes permettant d'exécuter le test.

Les jeux de test doivent être conçu au moment de la spécification du programme, ils permettent souvent de préciser le comportement attendu des programmes.

2.5 Primitives de base

Nous décrivons dans cette partie les primitives de base du langage de description (permettant de décrire les spécifications et les réalisations) et leur correspondance dans le langage de programmation (codage en CAML). Nous présentons en particulier :
– les expressions primitives (domaines de base, fonctions prédéfinies)
– les outils de composition (définitions par cas, composition de fonctions)
– les mécanismes d'abstraction (nommer des expressions intermédiaires)

Pour chaque nouvelle construction (du langage de description ou de programmation), on donne les règles de syntaxe (comment écrire l'expression), les règles de bonne formation (quelles sont les utilisations licites de l'expression) et enfin les règles d'évaluation (comment se calcule l'expression). Les règles de bonne formation et d'évaluation sont appelées *règles sémantiques* du langage.

2.5.1 Domaines et fonctions de base

Les domaines primitifs dans le langage de description sont `entier, réel, booléen, caractère, chaînes de caractères` représentant respectivement l'ensemble des entiers, des réels, des booléens, des caractères et des chaînes de caractères.

Transparent

Les valeurs associées aux domaines de base sont :
– `entier` : 0,1,2,…,-1,-2,…
– `réel` : 1.25, …
– `caractère` : 'a' …'z', '0','1' …
– `chaînes de caractères` : "Hello" "World"

Les fonctions primitives associées sont les opérateurs usuels :
– `entier` : +, -, *, `div`, `mod`, fonctions de comparaison $<, \leq, >, \geq$.
– `réel` : +, -, *, /, fonctions de comparaison $<, \leq, >, \geq$.
– `chaînes de caractères` : on écrira $s_1.s_2$ la chaîne de caractères obtenue par concaténation des chaînes s_1 et s_2.

Fin 2.22

Les fonctions de base bénéficient d'une syntaxe particulière conforme aux habitudes mathématiques. On écrit $x + 2 * y$ pour représenter l'expression $+(x, *(2, y))$.

Notations On pourra utiliser les notations mathématiques usuelles. Si A est un domaine qui possède un élément 0 et une fonction de comparaison $\geq$, on notera :

A^*	$= \{x : A \mid x \neq 0\}$	ensemble des éléments de A différents de 0
A_+	$= \{x : A \mid x \geq 0\}$	ensemble des éléments de A positifs ou nuls

On pourra ainsi écrire $\texttt{entier}_+$ pour les entiers positifs ou nuls et $\texttt{entier}^*_+$ pour les entiers strictement positifs.

On utilise également la notation $[n_1..n_2]$ pour représenter l'ensemble $\{k \in \mathbb{N} \mid n_1 \leq n \leq n_2\}$ des entiers compris entre n_1 et n_2.

Entiers en CAML

Le type primitif `int` de CAML pour les entiers ne correspond qu'à un sous-ensemble fini des entiers mathématiques (entre -2^{30} et $2^{30}-1$). Les fonctions arithmétiques sont effectuées modulo 2^{31}.

Transparent

```
# 1000000000;;
- : int = 1000000000
# 1000000000 + 1000000000;;
- : int = -147483648
```

Fin 2.23

Nombres réels en CAML

Le type `float` des nombres flottants représente un sous-ensemble des réels. Les fonctions arithmétiques suivent les conventions de la norme IEEE 754 avec 53 bits de mantisse et un exposant compris entre -1022 et 1023.

Il est possible si nécessaire de construire des programmes pour manipuler de grands entiers ou des rationnels en précision arbitraire et même des nombres réels "calculables". Une bibliothèque pour les rationnels en précision arbitraire existe pour CAML.

Ce cours n'abordera pas le problème de la précision des calculs numériques.

Dans le langage de description les mêmes symboles sont utilisés pour les fonctions sur les entiers et sur les réels. Ce n'est pas le cas dans CAML, les fonctions et les constantes sur les nombres flottants s'écrivent différemment de celles sur les entiers.

Transparent

```
# 1.0;;
- : float = 1
# 1;;
- : int = 1
# sqrt 2;;
This expression has type int but is here used with type float
# sqrt 2.;;
- : float = 1.41421356237
# 1.5+2;;
This expression has type float but is here used with type int
# 1.5+.2.;;
- : float = 3.5
```

Fin 2.24

Chaînes de caractères en CAML

Transparent

Le type des caractères se note `char`.

Les valeurs du type `char` sont `'a'` (le caractère `a`), `'\200'` (le caractère de code ASCII 200 : È), ...

Le type des chaînes de caractères se note `string`.

Les valeurs du type `string` sont "" (chaîne vide), "abab", ...

La Fonction de concaténation se note `^`.

```
# "Une chaîne"^" de caractères";;
- : string = "Une chaîne de caractères"
```

Fin 2.25

2.5.2 Le domaine des booléens

Nous décrivons plus particulièrement le domaine `booléen` qui joue un rôle important dans la caractérisation de sous-domaines.

Le domaine des booléens est composé des valeurs {`vrai`,`faux`}. Les fonctions de base sont les connecteurs logiques `et`, `ou` et la négation `non`.

Chaque fonction sur les booléens peut être définie par sa table de vérité, qui décrit le résultat de la fonction pour toutes les valeurs possibles des paramètres d'entrée (ceci est possible car les domaines des arguments sont finis)

Transparent

`vrai et vrai`	= `vrai`
`vrai et faux`	= `faux`
`faux et vrai`	= `faux`
`faux et faux`	= `faux`

`vrai ou vrai`	= `vrai`
`vrai ou faux`	= `vrai`
`faux ou vrai`	= `vrai`
`faux ou faux`	= `faux`

`non vrai`	= `faux`
`non faux`	= `vrai`

Fin 2.26

Booléens en CAML

Transparent

Le domaine des booléens se traduit en CAML par le type `bool`.

Les valeurs booléennes se notent `true` pour `vrai` et `false` pour `faux`.

La Fonction `et` se note `&`, la fonction `ou` se note `or` (ou `||`) et la fonction `non` se note `not`.

Fin 2.27

Les fonctions booléennes sont des fonctions dont le résultat est dans le domaine `booléen`, elles sont également appelées *prédicats*.

Égalité-inégalité L'égalité est un prédicat qui sert dans de nombreuses spécifications. Au niveau d'un langage de programmation, on dispose au minimum des fonctions de test d'égalité entre des objets primitifs (entiers, flottants, caractères ...). Certains langages, comme CAML, ont comme fonction primitive le test d'égalité entre deux objets quelconques. Ce qui est testé est l'égalité de la représentation machine des valeurs. Dans d'autres langages, il peut être nécessaire d'implanter soi-même la fonction de test d'égalité pour des objets complexes. De plus, il peut arriver que dans un certain domaine, on veuille identifier des objets qui n'ont pas la même représentation machine. Par exemple, lorsqu'on représente des nombres réels par des flottants, on peut identifier deux flottants dont la différence est petite.

Dans notre langage de description, on notera $e_1 = e_2$ l'expression booléenne qui vaut `vrai` lorsque e_1 et e_2 calculent la même valeur. Le profil de l'égalité est $(t; t) \rightarrow$ `booléen`.

De même on introduit un prédicat testant l'inégalité : l'expression booléenne $e_1 \neq e_2$ vaut `vrai` lorsque e_1 et e_2 calculent des valeurs différentes. Cette expression est équivalente à `non`$(e_1 = e_2)$.

Égalité-inégalité en CAML En CAML, l'expression qui teste si deux objets a et b sont égaux (resp. différents) se note a=b (resp. a<>b).

Sous-ensemble Un prédicat de profil $t \rightarrow$ `booléen` représente un ensemble de valeurs du domaine associé à t à savoir l'ensemble des valeurs pour lesquelles la fonction renvoie `vrai`.

2.5.3 Expressions conditionnelles

La modélisation de fonctions nécessite souvent de définir des expressions en distinguant différents cas suivant les données du problème. On parle alors d'expressions conditionnelles.

Transparent

Syntaxe du langage de description Une conditionnelle s'écrit :

`si` *cond* `alors:` $expr_1$ `sinon:` $expr_2$

Condition de bonne formation Le résultat de l'évaluation de l'expression *cond* doit appartenir au domaine `booléen` et les expressions $expr_1$ et $expr_2$ doivent donner comme résultat de leur évaluation une valeur appartenant au même domaine t.

Syntaxe CAML Son codage en CAML est direct :

`if` *codage(cond)* `then` *codage(*$expr_1$*)* `else` *codage(*$expr_2$*)*

Condition de bonne formation L'expression *codage(cond)* doit être de type `bool` et les expressions *codage(*$expr_1$*)* et *codage(*$expr_2$*)* doivent avoir le même type t, t est alors le type de l'expression conditionnelle complète.

Fin 2.28

Transparent

Valeur Si le résultat de l'évaluation de *cond* est `vrai` (resp. `faux`) et le résultat de l'évaluation de $expr_1$ (resp. $expr_2$) est v alors le résultat de l'évaluation de `if` *cond* `then` $expr_1$ `else` $expr_2$ est v

Exemples La fonction `min` peut s'écrire en langage de réalisation puis en CAML par :

$$\texttt{min} : (x, y) \mapsto \textbf{si } x < y \textbf{ alors} : x \textbf{ sinon} : y$$

```
let min x y = if x < y then x else y;;
```

Fin 2.29

2.5.4 Mécanismes d'abstraction

L'abstraction permet de donner un nom à une expression intermédiaire. Elle sert à :
- structurer une expression en identifiant des sous-expressions,
- éviter d'écrire et d'évaluer plusieurs fois la même expression.

L'abstraction permet d'associer simultanément plusieurs noms à plusieurs sous-expressions.

Mécanisme d'abstraction dans le langage de description.

Il s'agit d'un mécanisme d'abstraction globale dont la portée est l'expression définissant le corps d'une fonction dans la réalisation.

Transparent

Syntaxe de l'abstraction dans le langage de description:

$$\texttt{soit}\, id_1 = e_1\, , ..., id_p = e_p,\, e$$

Les noms id_1, ...,id_p sont des identificateurs pour les expressions e_1, ..., e_p auxquels ils sont liés. Ils peuvent être utilisés dans toutes les expressions qui les suivent dans le corps d'une fonction donnée. Dans chacune de ces expressions, ils remplacent la valeur de l'expression qui leur est associée.

Fin 2.30

Transparent

Exemple:

`min`(a,b) $\mapsto$	**soit**	x = abs(a-b),
		y = a+b,
	(x+y)/2	
`min4`(a,b,c,d) $\mapsto$	**soit**	x =`min`(a,b),
		y =`min`(x,c),
	`min`(y,d)	

Fin 2.31

Transparent

Règles de bonne formation de l'abstraction dans le langage de description: Ne pas redéfinir plusieurs fois le même identificateur dans une même définition de fonction.

Valeur: L'évaluation d'une expression $\texttt{soit}\, id_1 = e_1 ..., id_p = e_p,\, e$ consiste à remplacer chaque occurrence de chaque identificateur id_i par le résultat de l'évaluation de l'expression e_i à laquelle il est lié, *en procédant dans l'ordre d'écriture des différentes liaisons*. Le résultat de l'évaluation est alors le résultat de l'évaluation de l'expression e ainsi dépliée (dans laquelle chaque identificateur a été remplacé par sa valeur).

Fin 2.32

Mécanismes d'abstraction dans CAML

Dans CAML, on distingue l'abstraction globale et l'abstraction locale. L'abstraction globale sert en fait à définir des fonctions en liant des identificateurs (les noms des fonctions) aux expressions les définissant.

Transparent

Syntaxe de l'abstraction globale dans CAML:

$$\texttt{let}\, id_1 = e_1\, ;;$$
$$...;;$$
$$\texttt{let}\, id_p = e_p;;$$

Exemple:

```
#let minv1 a b = (a + b - abs(a-b))/2 ;;

#let minv2 a b  = if a <= b then a else b ;;

#let min4 a b c d  = minv2 (minv2 a b) (minv2 c d) ;;
```

Fin 2.33

Dans une abstraction locale, on fixe l'expression dans laquelle les noms introduits peuvent être utilisés.

Transparent

Syntaxe CAML de l'abstraction locale:
Soient $id_1, \ldots id_p$ des identificateurs et $expr_1, \ldots expr_p, expr$ des expressions, on construit une nouvelle expression par abstraction locale :

$$\texttt{let } id_1 = expr_1 \texttt{ and } \ldots \texttt{ and } id_p = expr_p \texttt{ in } expr$$

Règles de bonne formation: le nom id_i, en tant que représentant de la valeur de $expr_i$ n'a de sens que dans l'expression *expr*. Les liaisons doivent être indépendantes entre elles : si le nom id_i apparaît dans $expr_j$ alors l'expression `let...in` ne sera bien formée que si id_i a été introduit dans un contexte englobant l'expression `let...in`.

Valeur: Le résultat de l'évaluation est alors le résultat de l'évaluation de l'expression e ainsi dépliée (dans laquelle chaque identificateur a été remplacé par sa valeur).

Exemple

$$\texttt{let } x = 2 \texttt{ and } y = 3 \texttt{ in } (f\ (x * y)\ (x + y))$$

est une expression équivalente à $(f\ (2 * 3)\ (2 + 3))$.

Fin 2.34

Transparent

Mise en garde: les règles de bonne formation d'une expression CAML `let ... in` sont différentes des règles d'abstraction du langage de description. En particulier, la portée des liaisons n'est pas la même: dans CAML, on peut lier différentes occurences du même identificateur à des expressions différentes. Il faut alors être très strict et rigoureux dans la définition de la portée de chaque occurrence de chaque identificateur.

Exemple On pourra écrire `min4` de la manière suivante :

$$\texttt{min4}\, a\, b\, c\, d\ = \texttt{let } x = \texttt{min}\, a\, b\ \texttt{ in let } y = \texttt{min}\, c\, x\ \texttt{ in min}\, d\, y$$

mais pas :

$$\texttt{min4}\, a\, b\, c\, d\ = \texttt{let } x = \texttt{min}\, a\, b\ \texttt{ and } y = \texttt{min}\, c\, x\ \texttt{ in min}\, d\, y$$

Dans une désignation comportant un `and`, les différentes expressions doivent être indépendantes.

Fin 2.35

2.6 Constructions avancées

Nous décrivons dans cette partie des constructions avancées pour spécifier, réaliser ou coder des calculs. Ces constructions sont spécifiques aux langages que nous étudions. Nous présentons en particulier :

– les conditionnelles multiples,
– les définitions par filtrage,
– les exceptions,
– les fonctions partielles.

2.6.1 Conditionnelle multiple

Peu de constructeurs d'expressions suffisent pour décrire tous les calculs exécutables par une machine. Cependant, cela aboutit à des expressions lourdes au dépens de la lisibilité de la réalisation. Dans le langage de description, on introduit donc des expressions non primitives qui peuvent s'exprimer en fonction d'expressions plus simples.

C'est le cas de la conditionnelle multiple.

Transparent

Conditionnelle multiple dans le langage de description: Soit $cond_1, \ldots cond_n, expr_1, \ldots, expr_n, expr$ des expressions et $x_1, \ldots, x_k$ l'ensemble des variables apparaissant dans $cond_1, \ldots cond_n$ alors :

$$\begin{array}{l} \textbf{selon le cas } x_1, \ldots, x_k \\ \quad cond_1 : expr_1 \\ \quad cond_2 : expr_2 \\ \quad \vdots \qquad : \\ \quad cond_n : expr_n \\ \quad \texttt{sinon}: expr \end{array}$$

est une expression qui est équivalente à l'expression :

$$\begin{array}{lll} \textbf{si } cond_1 \textbf{ alors}: expr_1 & & \\ \textbf{sinon}: & \textbf{si } cond_2 \textbf{ alors}: expr_2 & \\ & \vdots & \\ & \textbf{sinon}: & \textbf{si } cond_n \textbf{ alors}: expr_n \\ & & \textbf{sinon}: expr \end{array}$$

Fin 2.36

Les règles de bonne formation et d'évaluation de cette expression se déduisent de l'équivalence.

Dans le cas où les conditions $cond_i$ couvrent l'ensemble des cas, la clause `sinon:` peut être omise. On écrira :

$$\begin{array}{l} \textbf{selon le cas } x_1, \ldots, x_k \\ \quad cond_1 : expr_1 \\ \quad cond_2 : expr_2 \\ \quad \vdots \qquad : \\ \quad cond_n : expr_n \end{array}$$

qui est équivalent à l'expression :

$$\begin{array}{lll} \textbf{si } cond_1 \textbf{ alors}: expr_1 & & \\ \textbf{sinon}: & \textbf{si } cond_2 \textbf{ alors}: expr_2 & \\ & \vdots & \\ & \textbf{sinon}: & \textbf{si } cond_{n-1} \textbf{ alors}: expr_{n-1} \\ & & \textbf{sinon}: expr_n \end{array}$$

Exemple

L'exemple du minimum se réécrit plus simplement par cas :

Transparent

$$\texttt{min} : (a, b) \mapsto \textbf{si } a < b \textbf{ alors} : a \textbf{ sinon} : b$$

La fonction `min4` peut s'écrire directement à l'aide de la construction `selon le cas`.

$$\begin{array}{ll} \texttt{min4} : (a, b, c, d) \mapsto & \textbf{selon le cas } a, b, c, d \\ & \quad a \leq b \texttt{ et } a \leq c \texttt{ et } a \leq d \text{: } a \\ & \quad b < a \texttt{ et } b \leq c \texttt{ et } b \leq d : b \\ & \quad c < a \texttt{ et } c < b \texttt{ et } c \leq d : c \\ & \quad d < a \texttt{ et } d < b \texttt{ et } d < c \text{: } d \end{array}$$

Fin 2.37

On se convint simplement de la correction de cette expression. Il est plus délicat de se convaincre que celle-ci est complète, pour cela il suffit de remarquer que la première condition couvre tous les cas où a est minimum, la seconde tous les cas où b est minimum et a ne l'est pas, la troisième tous les cas où c est minimum mais ni a ni b ne le sont ....

Cette réalisation n'est pas efficace car elle fait toujours au moins trois tests et peut dans le cas le pire où elle teste toutes les conditions faire neuf tests (si trois conditions sont fausses la troisième est vérifiée). Alors que la fonction `min` ne fait qu'un test et que `min4` peut se réaliser à l'aide de 3 fonctions `min`.

Remarque On aurait pu choisir la conditionnelle multiple comme primitive et introduire la conditionnelle simple comme un cas particulier. Cependant on retrouve la conditionnelle simple dans tous les langages de programmation alors que la conditionnelle multiple telle que nous la présentons ici n'existe pas dans la plupart des langages de programmation. Pour des raisons de lisibilité et d'efficacité, les langages offrent cependant des primitives de choix multiple, mais dont le comportement varie d'un langage à l'autre. Dans le langage CAML que nous utiliserons, les définitions par filtrage permettent de définir une expression par cas suivant des sous-domaines. Mais au lieu de représenter ces sous-domaines par des prédicats, CAML utilise des expressions particulières appelées *motifs*, qui offrent des notations particulièrement élégantes dans le cas des définitions de types structurés.

2.6.2 Filtrage

CAML ne possède pas de conditionnelle multiple traduisant la construction `selon le cas` de notre langage de description. Par contre CAML propose une notation générale qui permet de raisonner par cas suivant la forme de la valeur d'une expression.

$$\begin{array}{lll} \texttt{match } \mathit{expr} \texttt{ with} & \mathit{motif}_1 & \rightarrow \mathit{expr}_1 \\ & \mid \mathit{motif}_2 & \rightarrow \mathit{expr}_2 \\ & \vdots & \\ & \mid \mathit{motif}_n & \rightarrow \mathit{expr}_n \end{array}$$

Chaque *motif* représente un ensemble de valeurs possibles pour l'expression. À chaque motif, est associé un prédicat $\mathit{expr} \in \{\mathit{motif}\}$ qui définit à quelle condition l'expression *expr* représente une valeur compatible avec la motif *motif*. On dit que l'expression `match` *expr* `with` $\mathit{motif}_1 \rightarrow \mathit{expr}_1 | \ldots | \mathit{motif}_n \rightarrow \mathit{expr}_n$ est construite par *filtrage* de *expr* selon les motifs motif_i.

Cette expression peut se voir comme une notation élégante pour une conditionnelle multiple :

$$\texttt{if}\ expr \in \{motif_1\}\ \texttt{then}\ expr_1\ \texttt{else}\ldots\texttt{if}\ expr \in \{motif_n\}\ \texttt{then}\ expr_n\ \texttt{else}\ erreur$$

Les règles de typage de cette expression est que chaque *expr* doit être bien typée d'un type t, que chaque motif doit représenter un ensemble de valeurs du type t et toutes les expressions $expr_i$ doivent être du même type t' qui est également le type de l'expression tout entière.

Interprétation des motifs Pour l'instant nous ne verrons que des motifs très simples, nous introduirons des motifs plus complexes lorsque nous introduirons les types structurés de CAML.

Les motifs que nous considérons sont soit des constantes d'un type de base, soit le caractère spécial de soulignement `_`.

L'ensemble de valeurs représenté par une constante contient juste cette valeur. L'ensemble des valeurs représenté par le caractère spécial `_` représente toutes les valeurs du type de l'expression que l'on est en train de filtrer.

C'est-à-dire que $e \in \{c\}$ est équivalent à $e = c$ et $e \in \{\_\}$ est équivalent à `true`. On utilise le motif `_` pour couvrir tous les cas non-traités par les motifs précédents.

Transparent

Exemples

```
let negate b = match b with
    true  -> false
  | false -> true;;

let is_zero n = match n with
    0     -> true
  | _     -> false;;
```

Fin 2.38

Filtrage sur plusieurs expressions Il est possible d'utiliser la construction `match` décrite pécédemment pour discriminer suivant la forme de plusieurs expressions. L'expression à filtrer est alors remplacée par un n-uplet d'expressions et chaque motif par un n-uplet de motifs.

Transparent

$$\begin{array}{lll}\texttt{match}\ (e_1,\ldots,e_k)\ \texttt{with} & (m_1^1,\ldots,m_1^k) & \to expr_1 \\ & \mid (m_2^1,\ldots,m_2^k) & \to expr_2 \\ & \vdots & \\ & \mid (m_n^1,\ldots,m_n^k) & \to expr_n\end{array}$$

Fin 2.39

On détermine la branche en regardant l'appartenance de chaque expression e_i à l'ensemble associé au motif m_j^i correspondant.

L'expression précédente aura la même valeur que $expr_1$ si $e_1 \in \{m_1^1\}$ et $e_2 \in \{m_1^2\}$ et $\ldots e_k \in \{m_1^k\}$. Plus généralement, c'est une abréviation pour l'expression :

$$\begin{array}{l}\texttt{if}\ e_1 \in \{m_1^1\}\ \&\ e_2 \in \{m_1^2\}\ \&\ \ldots\ \&\ e_k \in \{m_1^k\}\ \texttt{then}\ expr_1 \\ \texttt{else if}\ e_1 \in \{m_2^1\}\ \&\ e_2 \in \{m_2^2\}\ \&\ \ldots\ \&\ e_k \in \{m_2^k\}\ \texttt{then}\ expr_2 \\ \vdots \\ \texttt{else if}\ e_1 \in \{m_n^1\}\ \&\ e_2 \in \{m_n^2\}\ \&\ \ldots\ \&\ e_k \in \{m_n^k\}\ \texttt{then}\ expr_n \\ \texttt{else}\ erreur\end{array}$$

Transparent

```
#let imply x y = match (x,y) with
    true,true  -> true
  | true,false -> false
  | false,true -> true
  | false,false -> true;;

#let imply x y = match (x,y) with
    true,true  -> true
  | true,false -> false
  | false,_    -> true;;

#let imply x y = if x then y else true;;
```

Fin 2.40

2.6.3 Exceptions

Il peut arriver qu'une fonction n'ait pas de résultat naturel dans certains cas. Si on ne peut garantir que l'entrée de la fonction sera telle que cette situation n'arrivera jamais, alors il faut prévoir le traitement de ce cas.

Dans la description du calcul on utilisera la construction `erreur` qui peut être mise dans un programme à la place de n'importe quelle expression. Si le calcul de la valeur de l'expression nécessite l'évaluation de `erreur` alors toute l'expression a pour valeur `erreur`. Il est souhaitable d'associer à la construction `erreur` un commentaire indiquant la nature de l'erreur.

Transparent

Exemple Supposons que l'on représente les mois de l'année par des chaînes de caractères et que l'on veuille implanter une fonction qui calcule le nombre de jours du mois.

SPÉCIFICATION:

{Domaines}
`mois` {Domaine des mois de l'année}
{Constantes}
`jan, fév, mars, avr, mai, juin, juil, août, sept, oct, nov, déc : mois`
{Fonctions}
`nb_de_jours` : (m:`mois`) →`entier` {associe à un mois m son nombre usuel de jours}

Fin 2.41

Transparent

RÉALISATION:

```
mois=chaînes de caractères
jan="Janvier" fév="Février" ... déc="Décembre"
nb_de_jours:m ↦
   selon le cas m
       m=jan ou m=mars ou m=mai ou m=juil
       ou m=août ou m=oct ou m=déc          : 31
      m=avr ou m=juin ou m=sept ou m=nov   : 30
      m=fev                                : 28
      sinon                                : erreur {m n'est pas un mois}
```

Fin 2.42

Exceptions en CAML

CAML possède un mécanisme d'exception sophistiqué. En pratique on se servira d'une primitive `failwith` qui prend en argument une chaîne de caractères.

L'évaluation de `failwith` *message* a pour effet d'arrêter le calcul et d'afficher le message.

Transparent

La fonction `nb_de_jours` pourra se coder de la manière suivante :

```
# let nb_de_jours m =
    match m with
    "Janvier" -> 31   | "Fevrier"  -> 28   | "Mars"      -> 31
  | "Avril"   -> 30   | "Mai"      -> 31   | "Juin"      -> 30
  | "Juillet" -> 31   | "Aout"     -> 31   | "Septembre" -> 30
  | "Octobre" -> 31   | "Novembre" -> 30   | "Décembre"  -> 31
  | _  -> failwith "l'argument n'est pas un mois"

val nb_de_jours : string -> int = <fun>
```

Fin 2.43

2.6.4 Fonctions partielles

De nombreuses fonctions que nous avons à manipuler ne sont pas définies sur toutes les valeurs d'un domaine de base. Il est donc utile de pouvoir décrire restreindre les domaines d'application; pour cela nous utiliserons les prédicats.

Dans la signature de la fonction, nous ferons apparaître la condition que les entrées sont supposées vérifier.

Transparent

Soit $p(x_1, \ldots, x_n)$ une expression booléenne portant sur les paramètres $x_1, \ldots, x_n$. La signature

$$f : (x_1 : d_1; \ldots; x_n : d_n)\{p(x_1, \ldots, x_n)\} \to d$$

indique que la fonction f attend des paramètres $x_1, \ldots, x_n$ pour lesquels l'expression $p(x_1, \ldots, x_n)$ s'évalue en vrai et retourne un résultat dans le domaine d.

Exemple La division sur les réels pourra être déclarée avec la signature :

$$/ : (p, r : \texttt{réel})\{r \neq 0\} \to \texttt{réel}$$

Fin 2.44

Lorsqu'on utilise une fonction partielle f dans une expression, il faut s'assurer que ses paramètres vérifient bien la propriété p. Si cela n'est pas garanti, il faut alors englober l'appel de f dans une conditionnelle qui teste la propriété.

2.7 Évaluation des programmes CAML

Un programme CAML se présente comme une suite d'expressions et de déclarations séparées par des doubles points-virgules. CAML controle et calcule le type de chaque expression entrée. Il évalue l'expression et affiche son type et sa valeur.

2.7.1 Règles d'évaluation des expressions

Transparent

Soit f une fonction définie par une déclaration :

$$\texttt{let}\ f\ x_1 \ldots x_n = e\texttt{;;}$$

$x_1, \ldots, x_n$ sont des paramètres formels et e est une expression qui définit le corps de la fonction et dans laquelle apparaissent les x_i.

Son évaluation est lancée par l'appel de f sur des paramètres réels qui sont des expressions:

$$(f\ e_1\ \ldots\ e_n)\texttt{;;}$$

Evaluation: Les expressions e_i sont évaluées en des valeurs v_i. Dans l'expression e, chaque paramètre formel est remplacé par la valeur correspondante. La valeur de l'expression ainsi obtenue est le résultat de l'évaluation de la fonction.

Fin 2.45

2.7.2 Détecter les erreurs

Le contrôle de type effectué par CAML permet de détecter certaines erreurs. Tester les fonctions introduites est une seconde étape pour identifier des problèmes. Comme les fonctions peuvent en général s'appliquer à un nombre infini de données, il n'est pas possible de tester toutes les configurations. Il faut donc également justifier par une preuve que la fonction décrite réalise bien la tâche souhaitée. Cette étape de preuve se fait plutôt au niveau de la réalisation dans le langage de description.

2.7.3 Portée des identificateurs

Le nom d'une variable (identificateur) n'a de sens que par rapport à une expression où il apparaît :

- soit comme paramètre dans la définition d'une fonction
- soit comme identificateur dans une définition `let ...in`

Un même identificateur peut être utilisé plusieurs fois comme nom de variables dans un programme (par exemple x et y dans le programme précédent).

Il est important de rattacher sans ambiguïté une utilisation de variable à sa définition

Transparent

Bloc de portée d'un identificateur

Chaque définition correspond à une zone de texte (un *bloc*) où les variables introduites ont un sens :

Dans l'expression `let` $x = e_1$ `in` e_2, l'occurence introduite de x n'a de sens que dans l'expression e_2.

x peut apparaître dans un autre bloc, il prendra alors un autre sens (c'est-à-dire une autre valeur).

$$\texttt{let}\ x = e_1\texttt{;;}$$

$$\boxed{\begin{array}{l}\texttt{let}\ f\,x = \boxed{e}\texttt{;;}\\ \vdots\end{array}}$$

La première déclaration de x reste visible dans toute la suite du programme. Le paramètre x de f est visible dans l'expression e. Il cache la valeur précédente de x.

Fin 2.46

Transparent

```
#let x=7;;
x : int = 7
#let f z=z*x;;
f : int -> int = <fun>
#let x=true;;
x : bool = true
#(f 3);;
- : int = 21
```

La valeur de x dans la définition de f correspond à la valeur visible au moment de la déclaration de f. Le résultat de f n'est pas modifié par la redéfinition ultérieure de x.

Fin 2.47

Transparent

Cours n^o 3
Définition récursive de fonctions

Les primitives que nous avons introduites permettent de composer des fonctions, de sélectionner les opérations suivant des conditions sur les entrées. Pour résoudre des problèmes intéressants il est essentiel de pouvoir itérer les opérations un nombre de fois dépendant des données. En S1 et S2 vous avez vu ce comportement avec des boucles "for" et "while". En S3, ce comportement va être obtenu dans notre langage de description en définissant des fonctions *récursives*. La récursivité est étroitement liée à la notion de raisonnement par récurrence des mathématiques.
Définition: Une fonction f est dite *récursive* si le corps de la définition fait référence à f.

Fin 3.0

Transparent

Importance d'une analyse rigoureuse du problème:
- débouchant sur une **équation de récurrence** mathématiquement fondée
- permettant de prouver les propriétés de **terminaison** , de **complétude** et **complexité** des processus engendrés

Illustration sur la fonction factorielle: qui à un entier strictement positif n associe le produit des n premiers entiers positifs (noté $n!$).

Spécification $\texttt{fact}(n:\texttt{entier}^*_+) \rightarrow \texttt{entier}^*_+$ $\{\texttt{fact}(\text{n})$ calcule $n!\}$

Analyse

$$\texttt{fact}(n) = n \times (n-1) \times \ldots \times 1$$

Si $n = 1$ alors $\texttt{fact}(n) = 1$ si $n > 1$ alors l'équation suivante est vérifiée :

$$\texttt{fact}(n) = n \times \texttt{fact}(n-1)$$

La réalisation découle directement de l'analyse récurrente:

$$\texttt{fact} : (n) \mapsto \textbf{si } n = 1 \textbf{ alors} : 1 \textbf{ sinon} : n * \texttt{fact}(n-1)$$

Fin 3.1

3.1 Exemples

3.1.1 Puissance

Transparent

Problème Définir la fonction puissance qui à un entier positif ou nul n et un réel a associe la puissance n-ième de a (notée a^n)

Spécification $\texttt{puis}(a:\texttt{réel}; n:\texttt{entier}_+) \rightarrow \texttt{réel}$ $\{\texttt{puis}(\text{a,n})$ calcule a^n, $a^0 = 1\}$

Analyse

$$\mathtt{puis}(a,n) = \underbrace{a \times \ldots \times a}_{n \text{ fois}} = a \times \mathtt{puis}(a,n-1)$$

L'équation précédente est vérifiée lorsque $n \geq 1$. On a un cas particulier : $\mathtt{puis}(a,0) = 1$

Réalisation

$$\mathtt{puis} : (a,n) \mapsto \textbf{si } n = 0 \textbf{ alors} : 1 \textbf{ sinon} : a * \mathtt{puis}(a,n-1)$$

Fin 3.2

3.1.2 Ensemble de Cantor

Transparent

Problème Définir la fonction cantor qui étant donné un entier positif ou nul n, décide si, dans l'écriture de n en base 3 (écriture qui utilise les chiffres 0,1 et 2), n s'écrit en utilisant uniquement des 0 et des 1.

Spécification `cantor:entier`$_+$ $\rightarrow$ `booleen` {`cantor`(n) est vraie si n s'écrit en base 3 avec les chiffres 0 et 1 uniquement.}

Analyse

```
si (n mod 3)=2 alors cantor(n)= faux sinon cantor(n) = cantor(n div 3)
```

L'équation précédente est vérifiée lorsque $n \geq 1$. On a un cas particulier : $\mathtt{cantor}(0) = \mathtt{vrai}$

Réalisation

$$\mathtt{cantor} : (n) \mapsto \textbf{si } n = 0 \textbf{ alors} : vrai \textbf{ sinon} : \textbf{si } n \mathbin{\mathtt{mod}} 3 = 2 \textbf{ alors} : faux \textbf{ sinon} : \mathtt{cantor}(n \mathbin{\mathtt{div}} 3)$$

Fin 3.3

3.1.3 Autres exemples

Transparent

On retrouve le même mécanisme lorsqu'on cherche à caractériser une suite entière :

$$u_0 = a \qquad u_n = f(u_{n-1}) \text{ pour } n > 0$$

Récursivité double Il peut y avoir plusieurs appels récursifs :

$$\mathtt{fib}(n) = \mathtt{fib}(n-1) + \mathtt{fib}(n-2) \text{ pour } n > 1 \;\; \mathtt{fib}(0) = 0 \;\; \mathtt{fib}(1) = 1$$

Récursivité indirecte (croisée) Une fonction f peut faire référence à une fonction g qui elle-même fait référence à f. On dit que les fonctions sont mutuellement récursives.

Fin 3.4

Transparent

3.2 Principes d'une définition récursive

On trouve dans toute définition récursive *plusieurs cas* que l'on peut regrouper en deux *catégories*
- les cas *de base* (non récursifs)
- les cas généraux qui renvoient à la définition en cours (cas récursifs)

Analyse récurrente Le principe de toute analyse récurrente consiste à identifier ces différents cas, à s'assurer que la définition est bien formée en vérifiant que les cas généraux finissent par renvoyer à des cas de base.

Fin 3.5

En général on essaie d'identifier d'abord les cas les plus généraux puis les cas qui posent problèmes.

Transparent

On vérifie que tous les cas sont bien couverts (*complétude*) que l'on ne risque pas de boucler sur des cas généraux (*terminaison*).

Par exemple, pour la fonction de Fibonacci, si on oublie de poser `fib(1)=1`, on a un problème de non-complétude (cas $n = 1$ non couvert); si on définit

$$\mathtt{fib}(n) = \mathtt{fib}(n-1) + \mathtt{fib}(n-2) \text{ pour } n \geq 1 \quad \mathtt{fib}(0) = 1,$$

on a un problème de non-terminaison avec appels à $\mathtt{fib}(-1), \mathtt{fib}(-2), \ldots$.

Fin 3.6

Exemple de la factorielle Le cas général est $\mathtt{fact}(n) = n \times \mathtt{fact}(n-1)$ dans le cas où $n = 1$, on a $n - 1 = 0$ donc on sort de la spécification de `fact`, on décrit ce cas à part.

La complétude est assurée car tout entier positif est soit strictement plus grand que 1 et alors $\mathtt{fact}(n)$ est défini par le cas général, soit égal à 1 et $\mathtt{fact}(n)$ est décrit par le cas particulier.

La terminaison est assurée car $\mathtt{fact}(n)$ fait appel à $\mathtt{fact}(n-1)$ qui fait appel à $\mathtt{fact}(n-2)$ ...on a donc une suite d'appels sur des valeurs décroissantes et bornées par 1.

Exemple de la fonction de Cantor La complétude est assurée car tout entier positif est soit strictement plus grand que 0 et alors $\mathtt{cantor}(n)$ est défini par le cas général, soit égal à 0 et $\mathtt{cantor}(0)$ est décrit par le cas particulier. La terminaison est assurée car $\mathtt{cantor}(\mathrm{n})$ fait appel à $\mathtt{cantor}(n \ \mathtt{div}\ 3)$ qui fait appel à $\mathtt{cantor}((n \ \mathtt{div}\ 3) \ \mathtt{div}\ 3) \ldots$

On a donc une suite d'appels sur des valeurs strictement décroissantes et bornées par 0.

3.2.1 Terminaison

Transparent

Il faut s'assurer que chaque cas général renvoyant à la définition en cours le fait pour un cas *plus simple* que celui qui est considéré.

La terminaison est assurée si l'on peut montrer qu'à force de considérer des cas de plus en plus simples, on tombe forcément sur l'un des cas de base. Les cas de base constituent donc les *conditions d'arrêt* de la récursion.

On peut généralement définir une relation d'ordre entre les différents cas à considérer. Les renvois successifs correspondent alors à des cas strictements décroissants pour cette relation d'ordre.

Fin 3.7

Transparent

3.2.2 Réalisation d'une fonction récursive

La réalisation d'une fonction récursive revient toujours à une analyse par cas dans laquelle on distingue les cas généraux des cas particuliers.

Fin 3.8

3.2.3 Codage en CAML d'une fonction récursive

Transparent

La définition d'une fonction récursive nécessite l'emploi d'un nouvel opérateur d'abstraction `let rec`.

Exemple

```
#let rec fact n  = if n=1 then 1 else n*(fact (n-1));;
val fact : int -> int = <fun>

#let rec cantor n  =
  if n=0 then true else if (n mod 3)=2 then false else cantor(n/3);;
val cantor : int -> boolean = <fun>
```

Fin 3.9

3.3 Processus engendré par une fonction récursive

Transparent

L'évaluation d'une fonction récursive (sur une donnée correspondant à une valeur particulière) engendre un processus qui va lui même engendrer un certain nombre de sous-processus correspondant à l'évaluation de la fonction (sur des données plus simples).

Le nombre de ces sous-processus dépend de la valeur d'appel de la fonction ayant engendré le processus : **le processus est dit récursif.**

Fin 3.10

Transparent

Si on cherche à évaluer $\mathtt{puis}(5,3)$ les appels suivants vont être effectués :

$$\begin{array}{llll}
\mathtt{puis}(5,3) = 5\times\ \mathtt{puis}(5,2) & & & \\
 & \mathtt{puis}(5,2) = 5\times\ \mathtt{puis}(5,1) & & \\
 & & \mathtt{puis}(5,1) = 5\times\ \mathtt{puis}(5,0) & \\
 & & & \mathtt{puis}(5,0) = 1
\end{array}$$

Fin 3.11

3.3.1 Ressources consommées

Transparent

A chaque appel de `puis`(a, n) pour $n > 0$ il faut mettre en attente l'opération $a *$ `puis`$(a, n-1)$, évaluer `puis`$(a, n-1)$, restaurer le contexte de l'opération suspendue, évaluer le produit $a *$ `puis`$(a, n-1)$.

La mémorisation des opérations suspendues en cascade nécessite l'utilisation de ressources mémoire. Ici l'espace mémoire requis pour l'exécution du processus croît de façon linéaire par rapport à n.

Le temps nécessaire à l'exécution des opérations lors de l'appel de `puis`(a, n) croît aussi de façon linéaire par rapport à n.

Fin 3.12

3.4 Raisonnement par récurrence

Lorsqu'on travaille sur des définitions récursives, on est souvent amené à établir des résultats en raisonnant par récurrence (preuves de terminaison notamment).

Une *preuve par récurrence* est une méthode de raisonnement qui vise à établir une propriété pour tous les éléments d'un ensemble.

Elle s'appuie sur une définition récursive de l'ensemble des éléments pour lequel on cherche à établir la propriété.

Un ensemble E est défini récursivement si on se donne

1. des valeurs de base $a_1, \ldots, a_n$ appartenant à l'ensemble E;
2. des opérations de constructions f telles que si $y_1, \ldots, y_k$ appartient à E alors $f(y_1, \ldots, y_k)$ est également un élément de E;
3. un principe de minimalité qui dit que l'ensemble E est le plus petit ensemble qui vérifie les deux conditions précédentes. C'est-à-dire que si F est un ensemble qui contient les valeurs de base $a_1, \ldots, a_n$ et qui est stable par les opérations de constructions de E (ie. si $y_1, \ldots, y_k$ appartient à F alors $f(y_1, \ldots, y_k)$ appartient à F) alors E est inclus dans F.

Maintenant si on veut montrer la propriété

$$\forall x \in E. P(x)$$

il suffit de montrer que $E \subseteq \{x \in E \mid P(x)\}$. Pour cela il suffit de montrer en utilisant la condition de minimalité que $\{x \in E \mid P(x)\}$ est un ensemble qui contient les valeurs de bases de E (c'est-à-dire $a_1, \ldots, a_n$) et est stable par les opérations de construction de E (toute opération f de construction de E vérifie: si $y_1, \ldots, y_k \in \{x \in E \mid P(x)\}$ alors $f(y_1, \ldots, y_k) \in \{x \in E \mid P(x)\}$).

Cela revient à montrer que la propriété P est vérifiée pour tous les cas de base de la définition de l'ensemble :

$$P(a_1), \ldots, P(a_n)$$

et que la propriété P est préservée par toute opération de construction f de E :

$$\text{pour tout } y_1, \ldots, y_k, \text{ si } P(y_1) \text{ et } \ldots \text{et } P(y_k) \text{ alors } P(f(y_1, \ldots, y_k))$$

3.4.1 Preuve par récurrence sur les entiers

Transparent

On s'intéresse au cas de l'ensemble $\mathbb{N}$ des entiers positifs ou nul. Cet ensemble admet une définition récursive :

- valeur de base : $0 \in \mathbb{N}$

– constructeur : si $n \in \mathbb{N}$ alors $n+1 \in \mathbb{N}$

Le principe de preuve par récurrence associé est : Soit $P(n)$ la propriété à établir :

– on montre que $P(0)$ est vérifié;
– on montre que pour tout $n \in \mathbb{N}$ si $P(n)$ est vérifié alors $P(n+1)$ est vérifié;
– on en déduit par récurrence que la propriété $P(n)$ est vrai pour tout $n \in \mathbb{N}$.

Fin 3.13

3.4.2 Exemple

Vous avez probablement vu l'exemple ci-dessous en maths en S1.

Problème:

Transparent

Montrer que la propriété $P(n)$ suivante est vraie pour tout n : $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

Preuve:

– La propriété est vraie pour $n=0$ car $\sum_{i=0}^{0} i = 0 = \frac{0(1)}{2}$
– Supposons $P(n)$ pour $n \in \mathbb{N}$, montrons $P(n+1)$:

$$\begin{aligned}\sum_{i=0}^{n+1} i = \sum_{i=0}^{n} i + (n+1) &= \frac{n(n+1)}{2} + n + 1 \quad \text{par hypothèse de récurrence} \\ &= \frac{(n+2)(n+1)}{2}\end{aligned}$$

– on en déduit par récurrence que $P(n)$ est vraie pour tout n

Fin 3.14

3.4.3 Autres formes de récurrence

À partir du schéma de récurrence de base sur les entiers, on peut déduire d'autres schémas.

Transparent

Récurrence double sur $\mathbb{N}$

– On montre $P(0)$ et $P(1)$;
– On montre pour tout $n \in \mathbb{N}$ que si $P(n)$ et $P(n+1)$ alors $P(n+2)$;
– On en déduit que $P(n)$ est vrai pour tout $n \in \mathbb{N}$.

La justification de ce principe est une preuve par récurrence simple de la propriété $Q(n)$ définie comme "$P(n)$ et $P(n+1)$".

Fin 3.15

Par exemple,on peut utiliser la récurrence double pour démontrer que la suite de Fibonacci satisfait

$$fib(n) = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{n+1} - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^{n+1}$$

pour tout $n \geq 0$.

Transparent

Récurrence généralisée sur $\mathbb{N}$
- On montre $P(0)$;
- On montre pour tout $n \in \mathbb{N}$, si $\forall k \in \mathbb{N}.k \leq n \Rightarrow P(k)$ alors $P(n+1)$
- On en déduit que $P(n)$ est vrai pour tout $n \in \mathbb{N}$.

La justification de ce principe est une preuve par récurrence simple de la propriété $Q(n)$ définie comme "$\forall k \in \mathbb{N}.k \leq n \Rightarrow P(n)$".

Fin 3.16

3.4.4 Principales difficultés

- Trouver le bon principe de récurrence.
- Trouver la bonne formulation de la propriété.
  Le résultat à montrer par récurrence peut être plus fort que la propriété finale à montrer ; selon les mots de Polya, c'est le "paradoxe de l'inventeur". Par exemple, considérons le problème de trouver rapidement la somme de deux nombres binaires x et y écrits sur n bits. On suppose que n est une puissance de 2. Partageons ces nombres en deux morceaux de $\frac{n}{2}$ bits chacun et faisons l'addition sur chaque morceau par récurrence: $x = 2^{\frac{n}{2}}x_1 + x_2$, $y = 2^{\frac{n}{2}}y_1 + y_2$, on peut additionner x_1 à y_1 et x_2 à y_2. Une difficulté est posée par la possible propagation de la retenue.
  Cette difficulté est résolue en renforçant le résultat à calculer: le problème modifié est: trouver rapidement, d'une part la somme de $f(x, y)$ deux nombres x et y écrits sur n bits, et d'autre part de trouver la somme $g(x, y)$ de 1+ deux nombres x et y écrits sur n bits. A partir de $f(x_1, y_1)$, $f(x_2, y_2)$, $g(x_1, y_1)$ et $g(x_2, y_2)$ il est assez facile de calculer $f(x, y)$ et $g(x, y)$.
- Trouver les bons cas de base
  (certaines propriétés ne sont vraies qu'à partir de 1).
- Bien amorcer la récursivité
  (certaines propriétés sont bien héréditaires mais ne sont pas vérifiées pour tous les cas de base)
  Parabole de l'échelle, pour monter il ne suffit pas de savoir passer d'un échelon à l'autre, il faut également être capable d'atteindre le premier barreau ...
- Ne pas oublier d'opérations de construction.

3.5 Analyse récurrente d'une fonction, principes méthodologiques

L'analyse récurrente conduit à la définition récursive d'une fonction.

Le principe est toujours le même, on procède à une analyse par cas sur le domaine de la fonction à définir afin d'identifier :
- des cas simples pour lesquels on est en mesure de donner une expression directe de la valeur de la fonction, indépendamment d'elle-même
- des cas plus généraux, pour lesquels la valeur de la fonction se décrit à l'aide d'un (ou de plusieurs) appel(s) récursifs(s) de la même fonction appliquée à des arguments plus petits.

Transparent

3.5.1 Recherche d'équations récursives

Pour identifier les cas les plus généraux, on peut chercher à exprimer la valeur de la fonction sous la forme d'équations récursives :

$$f(x) = g(x, f(h(x)))$$

Dans une telle équation :

- h représente la façon dont on passe de la donnée x à une donnée "plus petite" $h(x)$ sur laquelle on procède à l'appel récursif $f(h(x))$. Cette fonction doit vérifier $h(x) < x$ pour un ordre $<$ bien fondé (qui n'admet pas de suite infinie décroissante $\ldots u_n < \ldots u_1 < u_0$).
- g traduit la façon dont on utilise le résultat de l'appel récursif pour décrire la valeur de $f(x)$.

Fin 3.17

Exemple de la fonction factorielle

Transparent

Dans l'exemple de la fonction `fact` l'équation récursive est

$$\texttt{fact}(n) = n * \texttt{fact}(n-1)$$

On a donc $h(n) = n - 1$ et $g(n,m) = n * m$.

Fin 3.18

Transparent

Cas particuliers On essaie ensuite d'identifier les cas pour lesquels de telles équations posent problème:

- soit parce qu'elles donnent des résultats faux,
- soit parce qu'elles ne correspondent pas à des expressions bien formées ($h(x)$ n'est pas dans le domaine de f).

On traite ces cas à part, ce sont les cas de base.

Fin 3.19

Propriété de l'ordre sur les paramètres

Si la fonction $f(x)$ utilise l'équation récursive $f(x) = g(x, f(h(x)))$ alors le processus récursif va calculer les valeurs $f(x), f(h(x)), f(h^2(x)), \ldots$. Il faut s'assurer que pour tout x dans le domaine de f, la suite des $h^n(x)$ comportera un des cas de base.

3.5.2 Exemple de la fonction puissance

Transparent

Problème: Définir la fonction `puis` qui à un réel a et un entier positif ou nul n associe a^n.

Précédente analyse fondée sur l'équation récursive : $a^n = a * a^{n-1}$.

Autre analyse fondée sur l'équation récursive:
Si n est pair alors $a^n = (a^{\frac{n}{2}}) \times (a^{\frac{n}{2}})$ sinon $a^n = a * (a^{\frac{n}{2}}) \times (a^{\frac{n}{2}})$.

Dans le cas où $n = 0$, on a $\frac{n}{2} = n$ et l'appel récursif ne se fait pas sur un élément plus petit.

On obtient la réalisation suivante:

```
puis: (a,n) ->
             si  n = 0 alors: 1
             sinon
               soit k = puis(a,n div 2),
               si n mod 2 = 0 alors: k * k sinon: a * k * k
```

Fin 3.20

Transparent

Cette réalisation est plus efficace que la première, car elle effectue beaucoup moins de multiplications.

Codage en CAML :

```
let rec puis a n =
     if n = 0 then 1
     else let k = (puis a (n / 2))
          in if n mod 2 = 0 then k * k else a * k * k
```

Fin 3.21

Pour illustrer le lien entre programme récursif et raisonnement par récurrence, nous allons démontrer que ce programme calcule correctement la fonction puissance.

Soit a un réel fixé. Soit $P(n)$ la propriété "la fonction calcule correctement a^n". Montrons par récurrence que $P(n)$ est vraie pour tout entier $n \geq 0$.

- Cas $n = 0$: la fonction retourne 1, qui est bien égal à a^0.
- Cas général: soit $n \geq 1$. On suppose (hypothèse de récurrence) que pour tout $\ell, 0 \leq \ell \leq n-1$, la propriété $P(\ell)$ soit vraie. On raisonne ensuite par cas suivant si n est pair ou impait.
  - Si n est pair, $n = 2m$, la fonction calcule $puis(a,m)^2$. Comme $0 \leq m \leq n-1$, la propriété $P(m)$ est vraie, donc $puis(a,m) = a^m$, et donc la fonction retourne $(a^m)^2 = a^{2m} = a^n$.
  - Si n est impair, $n = 2m+1$, alors la fonction calcule $a \times puis(a,m)^2$. Comme $0 \leq m \leq n-1$, la propriété $P(m)$ est vraie, donc $puis(a,m) = a^m$, et donc la fonction retourne $a \times (a^m)^2 = a^{2m+1} = a^n$.

  Donc la fonction retourne bien a^n dans tous les cas et $P(n)$ est vraie.

Transparent

Il est essentiel d'utiliser une construction `soit` pour éviter que chaque appel récursif engendre 2 nouveaux processus.

Avec le **soit**, pour calculer `puis`$(a,4)$ il faut calculer `puis`$(a,2)$ lequel requiert le calcul de `puis`$(a,1)$ lequel requiert le calcul de `puis`$(a,0)$ lequel est égal à 1. On calcule successivement `puis`$(a,1) = a \times 1 \times 1 = a$ puis `puis`$(a,2) = a \times a = a^2$ puis `puis`$(a,4) = a^2 \times a^2 = a^4$.

Fin 3.22

Transparent

Comparons cette réalisation à celle qui n'utilise pas de `soit`:

```
puis: (a,n) ->
            si  n = 0 alors: 1
            sinon
              si n mod 2 = 0 alors: puis(a,n div 2)*puis(a,n div 2)
                                sinon: a * puis(a,n div 2)*puis(a,n div 2)
```

Regardons le déroulement des appels récursifs pour calculer a^4.

Sans le **soit**, pour calculer `puis`$(a, 4)$ il faut calculer `puis`$(a, 2)$ et `puis`$(a, 2)$. Ceci revient à lancer deux processus, au lieu d'un seul avec le `soit` qui permet de faire une seule fois le calcul de `puis`$(a, 2)$, puis de réutiliser le résultat k pour calculer $k \times k$ sans avoir besoin de refaire le calcul de k. De plus, le même problème se pose de nouveau lors du calcul de `puis`$(a, 2)$, qui lance deux processus qui vont tous deux faire le calcul de `puis`$(a, 1)$, et il y aura donc au total quatre processus chargés de calculer `puis`$(a, 1)$, et ainsi de suite.

Fin 3.23

3.5.3 Autres exemples

Transparent

SPÉCIFICATION:
`somme_1_sur_x` (x:`entier`$^*_+$) $\rightarrow$`réel`$^*_+$
{calcule la somme $\frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{x}$}

Fin 3.24

Transparent

RÉALISATION:
`somme_1_sur_x` : $(x) \mapsto$
si $x = 1$ **alors**: 1
sinon: $\frac{1}{x}+$ `somme_1_sur_x`$(x - 1)$

Fin 3.25

Transparent

Codage en CAML

```
#let rec somme_1_sur_x x =
     if x=1
        then 1.
        else 1./.(float_of_int x) +. (somme_1_sur_x (x-1));;
val somme_1_sur_x : int -> float = <fun>
```

Fin 3.26

Transparent

Test

```
# (somme_1_sur_x 1);;
- : float = 1
# (somme_1_sur_x 2);;
- : float = 1.5
# (somme_1_sur_x 3);;
- : float = 1.83333333333
# (somme_1_sur_x 4);;
- : float = 2.08333333333
# (somme_1_sur_x 4555);;
- : float = 9.00130624006
# (somme_1_sur_x 40000);;
- : float = 11.1738628979
```

Fin 3.27

On vérifie qu'on a bien une fonction croissante de x. Que vaut-elle lorsque x tend vers l'infini ? Vous l'avez vu en maths, $\lim_{n\to\infty}(1 + \frac{1}{2} + \cdots + \frac{1}{n}) = +\infty$.

Transparent

SPÉCIFICATION:

`mult_carre` (x:`entier`$^*_+$) $\rightarrow$`entier`$^*_+$
{calcule le produit $1^2 * 2^2 * \ldots * x^2$}

Fin 3.28

Transparent

RÉALISATION:

`mult_carre` : $(x) \mapsto$
si $x = 1$ **alors**: 1
sinon: $x * x *$ `mult_carre`$(x-1)$

Fin 3.29

Transparent

Codage en CAML

```
#let rec mult_carre x =
    if x=1
    then 1
    else x*x*(mult_carre (x-1));;
val mult_carre : int -> int = <fun>
```

Fin 3.30

Transparent

Test

```
# (mult_carre 1);;
- : int = 1
# (mult_carre 2);;
- : int = 4
# (mult_carre 3);;
- : int = 36
# (mult_carre 4);;
- : int = 576
# (mult_carre 5);;
- : int = 14400
# (mult_carre 6);;
- : int = 518400
```

Fin 3.31

Cette fonction croît très rapidement, puisque vous reconnaissez bien sûr le carré de la fonction factorielle.

Transparent

Schéma Général Pour coder une fonction calculant les valeurs successives d'une suite $(x_n)_{n\in\mathbb{N}^*}$ définie par une équation $x_n = g(n, x_{n-1})$

`Traiter_suite` : $(n) \mapsto$
 si $n = 1$ **alors**: *constante*
 sinon: $g(n, \texttt{Traiter\_suite}(n-1))$

- Dans le cas de `somme_1_sur_x` la constante *cte* est 1, la fonction g est définie par $g(n, m) = (1/n) + m$.
- Dans le cas de `mult_carre` la constante *cte* est 1, la fonction g est définie par $g(n, m) = n * n * m$.

Fin 3.32

3.6 Extraction de la racine carrée

Nous traitons de l'exemple du calcul d'une racine carrée approchée d'un nombre.

3.6.1 Spécification

Problème Étant donné un réel positif ou nul x trouver la racine carrée y de x, c'est-à-dire y vérifiant $y^2 = x$.

SPÉCIFICATION:	
`racine` (x:`réel`$_+$) $\rightarrow$ `réel`$_+$	{calcule $\sqrt{x}$ }

Remarque : Comme on travaille sur des réels codés en machine, on ne peut obtenir en général qu'une valeur approchée de cette racine ce qui nécessite de reformuler le problème. On cherche une valeur approchée, c'est-à-dire pour un ϵ arbitraire, y tel que $|y^2 - x| < \epsilon$

SPÉCIFICATION:	
`racine` (x:`réel`$_+$; ϵ:`réel`$^*_+$) $\rightarrow$ `réel`$_+$	$\{|(\texttt{racine}(x,\epsilon))^2 - x| < \epsilon\}$

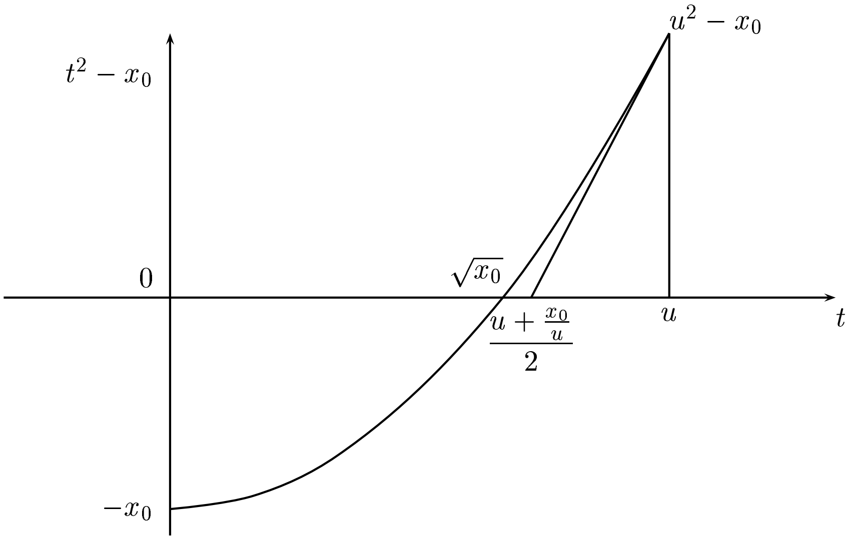
3.6.2 Analyse et Algorithme

La spécification caractérise le résultat attendu, il reste à fournir une méthode constructive pour calculer ces valeurs. Nous utiliserons une technique bien connue *la méthode de Newton.*

Méthode de Newton

La méthode de Newton s'applique à une fonction f sur les réels et permet de construire une suite qui converge vers un point t^* tel que $f(t^*) = 0$. On calcule les valeurs successives de cette suite jusqu'à trouver une estimation assez précise. Pour calculer la racine de x_0 il suffit de considérer la fonction $f(t) = t^2 - x_0$.

Transparent



Fin 3.33

Étant donné un point u de cette suite, le point suivant sera donné par l'intersection de l'axe des abscisses et de la tangente à la courbe de f en $(u, f(u))$.

Dans le cas de la fonction $f(t) = t^2 - x_0$, soit $u > 0$. La tangente en $(u, f(u))$ a pour pente $f'(u) = 2u$ et passe par le point $(u, u^2 - x_0)$. L'équation de la tangente est donc de la forme $d(t) = at + b = 2ut + b$ et vérifie $u^2 - x_0 = 2u \times u + b$ d'où $b = -u^2 - x_0$; cette tangente coupe l'axe des abscisses au point

$$u' = \frac{u^2 + x_0}{2u} = \frac{u + \frac{x_0}{u}}{2}$$

Justification Cette valeur u' est telle que

$$\sqrt{x_0} \leq u'$$

en effet $u' - \sqrt{x_0} = \dfrac{u^2 + x_0}{2u} - \sqrt{x_0} = \dfrac{(u - \sqrt{x_0})^2}{2u} \geq 0$.

De plus si $\sqrt{x_0} < u$ alors $u^2 + x_0 < 2u^2$ et donc $u' < u$, d'où $\sqrt{x_0} < u' < u$ et donc u' est une meilleure approximation que u de la racine de x_0.

On admettra que la suite $u_{n+1} = \dfrac{u_n + \frac{x_0}{u_n}}{2}$ converge (suite décroissante bornée) ; sa limite l vérifie $l \geq \sqrt{x_0}$ et $l = \dfrac{l^2 + x_0}{2l}$ la seule solution est $l = \sqrt{x_0}$.

Bilan Pour résoudre notre problème initial il faut :
- choisir une valeur initiale, par exemple 1,
- affiner l'estimation tant qu'elle n'est pas assez précise,
  - pour savoir si l'estimation courante u est assez précise, regarder si $|u^2 - x_0| < \epsilon$
  - pour affiner l'estimation u, calculer $\dfrac{u + \frac{x_0}{u}}{2}$

Ceci nous donne une décomposition de notre problème initial en sous-problèmes. Chaque sous-problème peut-être spécifié de façon autonome.

Transparent

Spécification

SPÉCIFICATION:

racine (x_0 : réel$_+$; ϵ : réel$^*_+$) $\rightarrow$ réel$_+$ {calcule u tel que $|u^2 - x_0| < \epsilon$}
assez_precis (x_0 : réel$_+$; ϵ : réel$^*_+$; u : réel$_+$) $\rightarrow$ booléen {renvoie vrai ssi $|u^2 - x_0| < \epsilon$}
affine_estime (x_0 : réel$_+$; u : réel$^*_+$) $\rightarrow$ réel$_+$ {calcule $\frac{u+\frac{x_0}{u}}{2}$}
cherche_estime (x_0 : réel$_+$; ϵ : réel$^*_+$; e : réel$^*_+$) $\rightarrow$ réel$_+$
{calcule u tel que $|u^2 - x_0| < \epsilon$ par la méthode de Newton à partir d'une estimation e. }

Fin 3.34

Transparent

Réalisation

RÉALISATION:

racine : $(x_0, \epsilon) \mapsto$ cherche_estime$(x_0, \epsilon, 1)$
assez_precis : $(x_0, \epsilon, u) \mapsto$ abs$(x_0 - u * u) < \epsilon$
affine_estime : $(x_0, u) \mapsto (u + (x_0/u))/2$
cherche_estime : $(x_0, \epsilon, e) \mapsto$ **si** assez_precis(x_0, ϵ, e) **alors**: e
sinon: cherche_estime$(x_0, \epsilon,$ affine_estime$(x_0, e))$

La terminaison des appels récursifs repose sur la preuve de convergence de la série.

Fin 3.35

Transparent

Codage en CAML

```
#let carre_float x = x *. x;;
val carre_float : float -> float = <fun>
#let affine_estim x u = (u +. (x /. u)) /. 2.;;
val affine_estim : float -> float -> float = <fun>
#let assez_precis x epsi u = abs_float (x -. (carre_float u)) < epsi;;
val assez_precis : float -> float -> float -> bool = <fun>
#let rec cherche_estim x epsi u =
  if (assez_precis x epsi u) then u
  else (cherche_estim x epsi (affine_estim x u));;
#let racine x epsi  = (cherche_estim x epsi 1.);;
val racine : float -> float -> float = <fun>
```

Fin 3.36

Transparent

Test

```
#let epsi = 0.0000001;;
val epsi : float = 1e-07
# (racine 4. epsi);;
- : float = 2
# (racine 2. epsi);;
- : float = 1.41421356237
```

Fin 3.37

Structurer le programme

Dans le codage précédent toutes les fonctions ont été mises au même niveau. Lors de l'analyse, les fonctions `affine_estime`, `cherche_estime` et `assez_precis` sont des fonctions auxiliaires qui sont apparues dans la décomposition du problème de calcul de `racine`. La fonction `carre_float` est par contre une fonction d'intérêt général qui pourra être réutilisée pour d'autres problèmes.

Cela nous amène à réécrire le codage CAML en ne rendant les définitions des fonctions `affine_estime`, `cherche_estime` et `assez_precis` visibles qu'à l'intérieur de la fonction `racine`. Les paramètres x_0 et ϵ ne varient pas au cours de la résolution du problème et sont fournis comme arguments à la fonction `racine`; il n'est donc pas nécessaire de les fournir explicitement comme argument aux fonctions `affine_estime`, `cherche_estime` et `assez_precis`.

On obtient alors le code suivant :

Transparent

```
# let racine x epsi =
  let affine_estim u  = (u +. (x /. u)) /. 2.
  in let assez_precis u  = (abs_float (x -. (carre_float u))) < epsi
  in let rec cherche_estim u  = if (assez_precis u) then u
                                else (cherche_estim (affine_estim u))
  in (cherche_estim 1.);;
val racine : float -> float -> float = <fun>
# racine 3. epsi;;
- : float = 1.73205081001
```

Fin 3.38

Transparent

Schéma général On aboutit à un schéma général de recherche de solution approchée :

`résoudre_un_problème`(*problème*, *solution_ approchée*) =
si `satisfaisante`(*solution_ approchée*) **alors**: *solution_ approchée*
sinon: `résoudre_un_problème`(*problème*,`améliorer`(*solution_ approchée*))

Fin 3.39

3.7 Séquences d'entiers

Une structure de données très utilisée est celle de séquences d'entiers. Une *séquence d'entiers* est un ensemble fini d'entiers placés dans un ordre précis.

3.7.1 Spécification

Le domaine des séquences d'entiers sera nommé `sequence_entiers`. Il comporte une constante `seq_vide` représentant la séquence d'entiers vide et une opération `ajout_debut` qui, étant donnés un entier n et une séquence s, construit la séquence comportant les mêmes éléments que s auxquels on a ajouté n au début.

Exemple La séquence formée des entiers 7 4 10 peut donc s'écrire :

soit $l =$ `ajout_debut`(7, `ajout_debut`(4, `ajout_debut`(10, `seq_vide`)))

Premier élément Un opération `premier` nous permet de sélectionner le premier élément d'une séquence non vide. On aura par exemple $\texttt{premier}(l) = 7$.

Reste de la séquence Une opération `reste` nous permet de calculer la séquence obtenue en retirant le premier élément d'une séquence non vide. On aura par exemple $\texttt{reste}(l) = \texttt{ajout\_debut}(4, \texttt{ajout\_debut}(10, \texttt{seq\_vide})$. De manière générale on a les égalités suivantes pour tout entier n et séquence l :

$$\texttt{premier}(\texttt{ajout\_debut}(n, l)) = n \qquad \texttt{reste}(\texttt{ajout\_debut}(n, l)) = l$$

Et si l est une séquence non vide alors :

$$l = \texttt{ajout\_debut}(\texttt{premier}(l), \texttt{reste}(l))$$

Spécification des séquences d'entiers

Transparent

SPÉCIFICATION:

{Domaines}
`sequence_entiers` {représente des séquences d'entiers}
{Constantes}
`seq_vide` {la séquence vide}
{Fonctions}
`ajout_debut`(n:`entier`;s:`sequence_entiers`) → `sequence_entiers`
{séquence obtenue en ajoutant n au début de s}
$\texttt{premier}(s : \texttt{sequence\_entiers})\ \{s \neq \texttt{seq\_vide}\} \rightarrow \texttt{entier}$
{premier élément de la séquence s}
$\texttt{reste}(s : \texttt{sequence\_entiers})\{s \neq \texttt{seq\_vide}\} \rightarrow \texttt{sequence\_entiers}$
{séquence s privée de son premier élément}

Fin 3.40

Transparent

SPÉCIFICATION:

{Propriétés}
$\texttt{premier}(\texttt{ajout\_debut}(n, s))=n$
$\texttt{reste}(\texttt{ajout\_debut}(n, s))=s$

Fin 3.41

3.7.2 Séquences d'entiers en CAML

CAML possède une bibliothèque définissant les opérations pour représenter des séquences d'entiers. Le domaine des séquences d'entiers correspond au type CAML appelé `int list`.

ATTENTION Pour avoir accès à la bibliothèque permettant de représenter les séquences il faut taper dans CAML la ligne suivante :

```
open List;;
```

Primitives pour les listes dans CAML

Nous donnons la correspondance entre les opérations du langage de réalisation et celles du langage de programmation.

Transparent

CAML	Langage de description
type `int list`	domaine `sequence_entiers`
`[]`	`seq_vide`
`_::_`	`ajout_debut`
n`::`l	`ajout_debut`(n, l)
`[`n_1`;`n_2`;`$\ldots$`;`n_p`]`	`ajout_debut`$(n_1,$ `ajout_debut`$(n_2, \ldots$ $\ldots$ `ajout_debut`$(n_p,$`seq_vide`$)\ldots))$
`[1;2;3]`	`ajout_debut(1,ajout_debut(2,ajout_debut(3,seq_vide)))`
`hd`	`premier`
`tl`	`reste`

Fin 3.42

Autres opérations

Beaucoup d'autres opérations sont utiles sur les séquences comme la mise bout à bout de deux séquences, le calcul de la longueur d'une séquence ou l'accès au n-ème élément de la séquence. Nous allons voir que toutes ces fonctions peuvent en fait se réaliser en utilisant un schéma récursif sur les séquences.

3.7.3 Analyse récurrente sur les séquences d'entiers

Caractérisation récursive des séquences d'entiers

Le domaine des séquences d'entiers est le plus petit ensemble E qui contient la séquence vide (`seq_vide` $\in E$) et tel que si $l \in E$ alors pour tout entier n on a aussi `ajout_debut`$(n, l) \in E$.

Fonction récursive sur les séquences Pour calculer une fonction f sur toutes les séquences il suffit de savoir calculer la valeur de f pour la séquence vide (cas de base) et pour une séquence s non vide, le calcul de $f(s)$ peut faire appel au calcul de $f($`reste`$(s))$. En effet `reste`(s) est une séquence strictement plus petite que s et en prenant des restes successifs, on est sûr d'arriver au cas de base de la séquence vide.

Analyse

Transparent

Lors de la phase d'analyse pour réaliser une fonction f de profil `sequence_entiers` $\rightarrow d$, si on choisit une analyse récurrente sur la liste, on cherchera à écrire des équations :

$$f(\texttt{seq\_vide}) = v \qquad f(s) = g(s, f(\texttt{reste}(s)))$$

où v et g dépendent du problème que f doit résoudre et s est non vide.

Fin 3.43

Pour trouver la fonction g, il est souvent utile d'écrire s sous la forme `ajout_debut`$($`premier`$(s),$`reste`$(s))$ et d'écrire $g(s, f($`reste`$(s)))$ sous la forme :

$$\textbf{soit } n = \texttt{premier}(s)\ ,\ s' = \texttt{reste}(s)\ ,\ g'(n, s', f(s'))$$

Lorsque l'analyse est faite, la traduction dans le langage de description et le codage sont immédiats :

Transparent

RÉALISATION:
$f : (s) \mapsto$ **si** $s =$ `seq_vide` **alors** : v **sinon** : $g(s, f(\texttt{reste}(s)))$

Codage en CAML :

```
let rec f s = if s=[] then v else (g s (f (tl s)))
```

Fin 3.44

Cas de fonctions à plusieurs arguments

Lorsqu'une fonction a plusieurs arguments, il faut choisir sur quel argument portera l'analyse récursive. Supposons que ce soit sur le premier qui est une séquence, les équations s'écriront :

$$f(\texttt{seq\_vide}, x) = v(x) \qquad f(s, x) = g(x, s, f(\texttt{reste}(s), x))$$

3.7.4 Fonctions usuelles sur les séquences

Tester si une séquence ne contient que des zéros

Il s'agit d'un exemple élémentaire de manipulation de séquence d'entiers : la récursivité esr terminale.

Transparent

SPÉCIFICATION:
`zeros` (s : `sequence_entiers`) $\rightarrow$ `booléen`   { vrai ssi la séquence s est vide ou ne contient que des zéros}

Analyse :
On sait qu'une séquence peut être soit vide soit construite par ajout d'un entier au début d'une séquence. Dans le premier cas, la fonction renvoie vraie (`zeros`(`seq_vide`)=vrai). Dans le second cas, si le premier entier de la séquence est non-nul, la fonction est fausse (`zeros`(s)=faux) ; s'il est nul, la fonction sera vraie ssi le reste de la séquence ne contient que des zéros (`zeros`(s)=`zeros`(reste(s))).

Fin 3.45

On en déduit le codage :

Transparent

Codage en CAML :

```
let rec zeros s =
    if s = [] then true else if (hd s)<>0 then false else (zeros (tl s));;
```

Fin 3.46

Longueur d'une séquence

Transparent

Spécification:
`longueur` (s : `sequence_entiers`) $\rightarrow$ `entier` { le nombre d'éléments de la séquence s}

Analyse :
On sait qu'une séquence peut être soit vide soit construite par ajout d'un caractère au début d'une séquence. Dans le premier cas, la longueur est 0 (`longueur(seq_vide)`=0). Dans le second cas, il y a exactement un entier de plus que dans la séquence à partir de laquelle elle a été construite (`longueur`(s)=1+ `longueur`(`reste`(s))).

Fin 3.47

On en déduit le codage :

Transparent

Codage en CAML :

```
let rec longueur s  =
    if s = [] then 0 else 1 + (longueur (tl s));;
```

Longueur en CAML La fonction primitive CAML `length` calcule la longueur d'une liste.

Fin 3.48

3.7.5 Concaténation de séquences

Transparent

Problème : Construire une fonction calculant la séquence obtenue en réunissant deux séquences bout à bout.

Spécification:
`concat` ($s1, s2$: `sequence_entiers`) $\rightarrow$ `sequence_entiers` { calcule la séquence formée des éléments de $s1$ suivis de ceux de $s2$}

Fin 3.49

Analyse :
Le type séquence étant défini récursivement, il faut procéder à une analyse récursive sur l'un des arguments. Les séquences étant construites par adjonction à gauche, il est plus judicieux de raisonner par rapport à la valeur du premier argument. On raisonne sur la façon dont a été construite la séquence s_1.

- Si s_1 correspond à la séquence vide, le résultat correspond à la seconde séquence s_2.

– Sinon s_1 a été construite par adjonction d'un élément à gauche d'une séquence. On essaie de satisfaire une équation récursive de la forme :

$$\texttt{concat}(s_1, s_2) = g(s_1, s_2, \texttt{concat}(\texttt{reste}(s_1), s_2))$$

Ici la fonction g correspond à l'ajout, en tête de la liste $\texttt{concat}(\texttt{reste}(s_1), s_2)$ du premier élément de s_1.

$$g(s_1, s_2, z) = \texttt{ajout\_debut}(\texttt{premier}(s_1), z)$$

Remarque : On vérifie que cette équation est bien typée, ce qui est toujours le cas car s_1 est supposée non vide.

Transparent

RÉALISATION:
`concat` : (s1,s2) $\mapsto$ **si** $s1$ = seq_vide **alors** : $s2$ **sinon** : `ajout_debut`(`premier`($s1$),`concat`(`reste`($s1$),$s2$))

Codage en CAML :

```
let rec concat s1 s2  =
        if s1 = [] then s2
        else let n= (hd s1) and l=(tl s1) in n::(concat l s2);;
```

Fin 3.50

Remarque : Cette opération est une opération prédéfinie sur les listes en CAML et correspond à l'opérateur infixe `@`. On peut donc représenter de façon équivalente le résultat de `concat(s1,s2)` par : `s1 @ s2`

3.7.6 Appartenance d'un élément à une séquence

Transparent

Problème : Construire une fonction qui vérifie si un entier n figure bien dans une séquence.

SPÉCIFICATION:
`appartient_seq` (e : `entier`; s : `sequence_entiers`) $\rightarrow$ `booléen` { vrai ssi $e \in s$}

Fin 3.51

Analyse :
Le type séquence étant défini récursivement, il faut procéder à une analyse récursive. On examine donc comment la séquence s a pu être construite.

– si s correspond à la séquence vide, le résultat est faux.

– si s n'est pas vide alors elle a été construite par adjonction d'un élément à gauche d'une séquence. On essaye de satisfaire une équation récursive de la forme :

$$\texttt{appartient\_seq}(e, s) = g(e, s, \texttt{appartient\_seq}(e, \texttt{reste}(s)))$$

Ici la fonction g doit renvoyer vrai si `appartient_seq`(e, `reste`(s)) est vrai. Sinon, la seule possibilité est que e corresponde à `premier`(s) soit :

$$g(e, s, b) = (e = \texttt{premier}(s)) \texttt{ ou } b$$

Transparent

Réalisation:
`appartient_seq` : $(e, s) \mapsto$ **si** $s =$ `seq_vide` **alors** : `faux` **sinon** : $e =$ `premier`(s) **ou** `appartient_seq`(e, `reste`(s)))

Codage en CAML :

```
let rec appartient_seq e s =
        if s = [] then false
        else let n= (hd s) and l= (tl s) in e = n or (appartient_seq e l);;
```

Fin 3.52

3.7.7 Inversion de séquence

Transparent

Problème : Construire une fonction qui construit la séquence inverse d'une séquence.

Spécification:
`inv_seq` (s : `sequence_entiers`) $\rightarrow$ `sequence_entiers` { calcule la séquence inverse de s}

Fin 3.53

Analyse :
Le type séquence étant défini récursivement, il faut procéder à une analyse récursive. On examine donc comment la séquence s a pu être construite.
– si s correspond à la séquence vide, le résultat est la séquence vide .
– si s n'est pas vide alors elle a été construite par adjonction d'un élément à gauche d'une séquence. On essaye de satisfaire une équation récursive de la forme :

$$\texttt{inv\_seq}(s) = g(s, \texttt{inv\_seq}(\texttt{reste}(s)))$$

Ici la fonction g doit pouvoir insérer `premier`(s) à la fin de la séquence `inv_seq`(`reste`(s)) soit :

$$g(x, y) = \texttt{ajout\_fin}(\texttt{premier}(x), y)$$

Mais cette fonction `ajout_fin` reste à construire. On a donc un nouveau (sous-)problème à résoudre.

Transparent

Problème : Construire une fonction qui ajoute un élément en fin d'une séquence.

SPÉCIFICATION:
`ajout_fin` (e : `entier`; s : `sequence_entiers`) $\rightarrow$ `sequence_entiers`
{ calcule la séquence obtenue en ajoutant l'élément e en fin de s}

Fin 3.54

Analyse :
Le type séquence étant défini récursivement, il faut procéder à une analyse récursive.

- si s correspond à la séquence vide, le résultat est la séquence ne contenant que l'élément e, *i.e.* `ajout_debut`(e, `seq_vide`)
- si s n'est pas vide alors elle a été construite par adjonction d'un élément à gauche d'une séquence. On essaye de satisfaire une équation récursive de la forme :

  $$\texttt{ajout\_fin}(e, s) = g(e, s, \texttt{ajout\_fin}(e, \texttt{reste}(s)))$$

  Ici la fonction g n'a qu'à rajouter `premier`(s) en tête de `ajout_fin`(e,`reste`(s)). On a donc $g(e, s, y) = \texttt{ajout\_debut}(\texttt{premier}(s), y)$.

Transparent

RÉALISATION:		
`inv_seq` : (s) $\mapsto$	**si** s = `seq_vide`	
	alors : s	
	sinon : `ajout_fin`(`premier`(s),`inv_seq`(`reste`(s)))	
`ajout_fin` : (e,s) $\mapsto$	**si** s = seq_vide	
	alors : `ajout_debut`(e,s)	
	sinon : `ajout_debut`(`premier`(s),`ajout_fin`(e,`reste`(s)))	

Fin 3.55

Transparent

Codage en CAML :

```
let rec ajout_fin e s =
        if s = [] then [e]
        else let n = (hd s) and l = (tl s) in n::(ajout_fin e l);;

let rec inv_seq s =
        if s = [] then []
        else let n = (hd s) and l = (tl s) in (ajout_fin n (inv_seq l));;
```

Remarque : Une autre façon de définir `ajout_fin` (qui revient au même) est de dire que $\texttt{ajout\_fin}(e, s) = \texttt{concat}(s, [e])$

Fin 3.56

3.7.8 Filtrage sur les séquences

Transparent

Le filtrage de CAML permet une écriture simplifiée des opérations de test et d'accès sur les séquences.

```
let rec f s =
     if s = [] then c
     else let n = (hd s) and l = (tl s) in (g n l (f l))
```

est équivalent à

```
let rec f s =
     match s with
        []     -> c
      | (n::l) -> (g n l (f l))
```

Fin 3.57

Dans le motif $n::l$, n et l sont de nouvelles variables.

Nous reviendrons sur la notion de séquence dans le chapitre 6 et de filtrage dans le chapitre 4.

Transparent

Cours n^o 4
Définition de types structurés

Les domaines de base vus (entier, réel, booléen, ...) sont insuffisants pour représenter des données structurées telles que les dates ou les matrices sauf à utiliser des codages peu naturels et parfois inefficaces. En particulier, toutes les fonctions que nous avons considérées dans les chapitres précédents ne renvoyaient comme résultats que des valeurs d'un domaine de base. Dans ce chapitre, nous étudions comment spécifier des domaines et des opérations pour représenter différentes structures (ensembles finis, n-uplets ...) et les utiliser comme arguments d'entrée ou de résultat des fonctions que nous définissons. Pour chaque structure, nous indiquons comment la coder en CAML.

Fin 4.0

4.1 Introduction

4.1.1 Exemple

Supposons que l'on veuille représenter un domaine `point` des points sur un écran. Un point est modélisé par son abscisse et son ordonnée qui peuvent être deux entiers x et y.

L'ensemble des couples d'entiers est dénombrable et il est donc possible de coder un point de coordonnées (x, y) par un entier (par exemple $2^x 3^y$). Cependant ce nombre ne présente pas d'intérêt. En particulier les entiers peuvent être additionnés, mais l'addition ne présente aucun intérêt ici puisque $2^x 3^y + 2^{x'} 3^{y'}$ n'a pas de sens dans ce contexte.

Au niveau du codage, si on mélange par erreur des objets de type point avec des objets de type entier, CAML ne s'en apercevra pas : on ne profite pas de la détection d'erreurs procurée par le typage fort de CAML.

Imaginons une représentation pour les points. Qu'est-ce que cette représentation doit absolument permettre de faire ? Ce qui est important est que l'on dispose de trois fonctions correspondant aux opérations de base sur le domaine représentant des points.

Transparent

- Une *opération de construction* (appelée par exemple `cons_point`) de signature :
  `cons_point`$(x, y$: `entier`$) \rightarrow$ `point` qui à partir de deux entiers x et y fournit un point (celui d'abscisse x et d'ordonnée y)
- deux opérations d'accès (appelées par exemple `abscisse` et `ordonnée`) de profil `point` $\rightarrow$ `entier` qui à un point associent respectivement son abscisse et son ordonnée.

Évidemment, ces deux opérations ne sont pas indépendantes, et on s'attend à ce que les égalités suivantes soient vérifiées :

$$\texttt{abscisse}(\texttt{cons\_point}(x, y)) = x \qquad \texttt{ordonnée}(\texttt{cons\_point}(x, y)) = y$$

Le domaine des points est un exemple de *domaine structuré* c'est-à-dire un domaine dont les valeurs sont décrites en composant des domaines plus simples.

Fin 4.1

4.1.2 Types abstraits de données

Les primitives offertes pour le codage de la représentation de domaines structurés peuvent être très variées d'un langage de programmation à l'autre (les listes sont primitives en LISP ou CAML mais nécessitent l'usage de pointeurs dans les langages impératifs). Même à l'intérieur d'un même langage de programmation, un domaine structuré peut être représenté de différentes manières (un couple de deux entiers peut se coder au moins de trois manières différentes en CAML).

Transparent

Il est donc souvent souhaitable de spécifier un domaine structuré et les opérations de construction et d'accès qui lui sont associées mais sans le réaliser dans le langage de description. Les autres opérations manipulant ce type ne dépendront donc pas de la représentation particulière choisie pour le domaine structuré. On parle alors de spécification de *type abstrait de données*, qui est ensuite directement codé à l'aide des primitives offertes par le langage de programmation considéré.

Fin 4.2

Dans ce chapitre nous modélisons au travers de différents types abstraits de données des domaines particuliers en spécifiant toutes les opérations utiles sur les valeurs de ces domaines. Pour chacun de ces domaines nous indiquons la manière de le coder dans le langage CAML. Les types structurés de CAML s'utilisent à travers les définitions par filtrage, les motifs qui seront utilisés sont plus riches que ceux que nous avons introduits pour les types primitifs.

4.2 Sous-domaines

4.2.1 Spécification

Certains domaines se modélisent simplement comme des sous-domaines de domaines existants. Par exemple, un jour peut se représenter comme un entier compris entre 1 et 31, un mois comme un entier de 1 à 12, une population par un entier positif représentant le nombre de milliers d'habitants, ...

Pour distinguer les différentes utilisations des entiers, la spécification introduit des noms différents pour chaque domaine :

SPÉCIFICATION:	
{Domaines}	
`jour`	{les jours de la semaine}
`mois`	{les mois de l'année}
`population`	{nombre d'habitants}

Il est préférable de ne pas préjuger a priori de la réalisation des domaines ci-dessus et d'éviter d'appliquer des opérations qui sont valides sur des entiers mais qui n'ont pas de sens lorsque ces entiers représentent des jours ou des semaines.

Cependant, si les domaines `jour`, `mois` ou `population` restent abstraits, il est pourtant nécessaire de revenir à une représentation sous forme d'un domaine connu (ici les entiers) afin de pouvoir effectuer des opérations concrètes. Ceci se fait simplement en introduisant deux opérations:

- une opération de construction de la représentation concrète vers le domaine abstrait,
- une opération d'accès qui étant donné un objet abstrait nous en donne une représentation concrète.

Évidemment ces deux opérations sont liées entre elles. L'accès à un objet construit à partir d'une valeur concrète, redonne cette valeur.

Forme générale Soit un domaine d. Soit une opération booléenne p sur ce domaine de profil $d \rightarrow$ `booléen`. On cherche à spécifier un domaine dont les valeurs peuvent être représentées par des objets v de d vérifiant la condition $p(v)$. La spécification d'un tel domaine se fera de la manière suivante :

Transparent

SPÉCIFICATION:	
{Domaine}	
dom	{sous-domaine équivalent à $\{x : d \mid p(x)\}$}
{Fonction de construction}	
$cons\_dom\ (x : d)\{p(x)\} \rightarrow dom$	
{Fonction de construction étendue}	
$d\_to\_dom\ (x : d) \rightarrow dom$	
{Fonctions d'accès}	
$proj : (x : dom) \rightarrow d$	

Les objets ainsi spécifiés vérifient de plus les propriétés :

$$\begin{array}{ll} proj(cons\_dom(x)) = x & \\ d\_to\_dom(x) = cons\_dom(x) & \text{si } p(x) \text{ est vrai} \\ d\_to\_dom(x) = \texttt{erreur} & \text{si } p(x) \text{ est faux} \end{array}$$

Fin 4.3

On distingue l'opération de construction $cons\_dom(x)$ qui ne peut être utilisée que si le contexte permet de montrer que x satisfait $p(x)$, de l'opération de construction étendue d_to_dom, qui accepte a priori n'importe quel x et se charge de vérifier qu'il est bien dans le sous-domaine associé à p.

4.2.2 Exemples

Transparent

Exemple 1 : `jour`

SPÉCIFICATION:	
{Domaine}	
`jour`	{jours vus comme des entiers de 1 à 31 }
{Fonction de construction}	
`cons_jour` (x:`entier`)$\{1 \leq x \leq 31\}$ →`jour`	
{Fonction de construction étendue}	
`entier_to_jour` (x:`entier`) →`jour`	
{Fonctions d'accès}	
`numéro_jour` (j:`jour`) →`entier`	

Fin 4.4

Transparent

Exemple 2 : `population`

SPÉCIFICATION:
{Domaine}
`population` {représente un nombre d'habitants en milliers }
{Fonction de construction}
`cons_popu` (x:`entier`)$\{0 \leq x\}$ →`population`
{Fonction de construction étendue}
`entier_to_popu` (x:`entier`) →`population`
{associe à n une population de n milliers d'habitants}
{Fonctions d'accès}
`nb_habitants` (p: `population`) →`entier`
{`nb_habitants`(p) est le nombre de milliers d'habitants de la population p}

Fin 4.5

4.2.3 Codage CAML

À partir d'un type primitif comme, le type `int`, CAML fournit la possibilité à l'utilisateur de construire de nouveaux types, dont les valeurs seront des entiers mais associés à une étiquette qui permet de leur attribuer un type de manière unique.

Ainsi on peut avoir deux copies des entiers, l'une associée à l'étiquette `Jour` qui nous donnera un objet de type `jour` et l'autre associée à l'étiquette `Mois` qui nous donnera un objet de type `mois`. Un objet dans le type `jour` ne pourra ainsi jamais être confondu avec un objet de type `mois` ou un objet de type `int`.

La déclaration de tels types se fait à l'aide de la commande

Transparent

Déclaration de types en CAML (1)

```
#type jour = Jour of int;;
#type mois = Mois of int;;
```

ou de manière plus générale

`type` *nouveau_ type* = *Etiquette* `of` *type*

Le nom de l'étiquette et celui du type peuvent être choisis librement par l'utilisateur. Cependant, OCaml impose que le nom de l'étiquette commence par une majuscule.

Fin 4.6

Transparent

L'étiquette agit comme une opération de construction :

```
#Jour(28);;
- : jour = Jour 28
#Mois(10);;
- : mois = Mois 10
```

Cependant ce type CAML ne permet pas de prendre en compte la restriction faite sur les objets du domaine.

```
#Jour(45);;
- : jour = Jour 45
```

Fin 4.7

Transparent

Pour construire des objets de type `jour` correspondant à des valeurs représentant des jours valides, c'est-à-dire à des entiers entre 1 et 31, on utilise une opération de construction qui teste que l'entrée est bien dans l'intervalle attendu et échoue lorsque cet entier ne correspond pas à un jour.

Si n est un jour valide on utilisera la fonction de construction:

```
#let cons_jour n = Jour(n);;
val cons_jour : int -> jour = <fun>
```

Si n est un entier quelconque on utilisera la fonction de construction étendue:

```
#let entier_to_jour n =
     if 1<=n & n<=31
        then Jour(n)
        else failwith "l'entrée n'est pas un jour valide";;
val entier_to_jour : int -> jour = <fun>
```

Fin 4.8

Accès

Transparent

Le mécanisme de filtrage de CAML permet de coder la fonction d'accès.

Si e est une expression de type `jour` et x un nouvel identificateur alors `Jour`(x) est un motif correct pour le type `jour`.

On peut donc écrire l'expression :

$$\texttt{match}\ e\ \texttt{with Jour}(x) \rightarrow f$$

La variable x introduite dans le motif est une nouvelle variable et a pour type `int`, sa portée est réduite à l'expression f. L'expression `match` a pour type t si f a pour type t.

Lorsque le programme e s'évalue en la valeur `Jour`(n) alors l'expression `match` précédente s'évalue comme l'expression `let` $x\ =\ n$ `in` f.

Dans le motif, x représente exactement l'entier associé à l'étiquette `Jour` pour former l'objet de type `jour`.

Fin 4.9

On peut donc simplement coder la fonction d'accès de la manière suivante:

Transparent

```
#let numero_jour j  = match j with Jour(x) -> x;;
val numero_jour : jour -> int = <fun>
```

On peut vérifier sur des exemples que `cons_jour` (resp. `entier_to_jour`) et `numero_jour` sont des opérations inverses l'une de l'autre :

```
#numero_jour(cons_jour 19);;
- : int = 19

#numero_jour(entier_to_jour 19);;
- : int = 19
```

Fin 4.10

Type étiqueté général On suppose introduit un nouveau type étiqueté :

Transparent

Type étiqueté général

$$\texttt{type}\ \mathit{nouveau\_type} = \mathit{Etiquette}\ \texttt{of}\ \mathit{type}$$

alors on peut construire l'expression CAML :

$$\texttt{match}\ e\ \texttt{with}\ \mathit{Etiquette}(x) \rightarrow f$$

La variable x introduite dans le motif est une nouvelle variable et a pour type *type*, sa portée est réduite à l'expression f. L'expression `match` a pour type t si e a pour type *nouveau_type* et f a pour type t.

Fin 4.11

Transparent

La fonction de construction a pour type CAML *type* $\rightarrow$ *nouveau_type* et se code à l'aide du constructeur :

$$\texttt{let}\ \mathit{cons\_dom}\ x = \mathit{Etiquette}(x)$$

Le constructeur étendu a pour type CAML *type* $\rightarrow$ *nouveau_type* et se code à l'aide du constructeur :

$$\texttt{let}\ \mathit{type\_to\_dom}\, x = \texttt{if}\ \ (p\ x)\ \ \texttt{then}\ \ \mathit{Etiquette}(x)\ \ \texttt{else failwith}\ \text{“}\mathit{erreur}''$$

L'opération d'accès a pour type *nouveau_type* $\rightarrow$ *type* et se code à l'aide du filtrage :

$$\texttt{let}\ \mathit{proj}\ a = \texttt{match}\ a\ \texttt{with}\ \mathit{Etiquette}(x) \rightarrow x$$

Lorsque la fonction de projection est définie, le filtrage devient superflu, en effet l'expression :

$$\texttt{match}\ e\ \texttt{with}\ \mathit{Etiquette}(x) \rightarrow f$$

est équivalente à `let` $x \ = \ (\mathit{proj}\ e)\ \mathit{in}\ f$

Fin 4.12

Remarques

- CAML ne permet pas de construire de représentation *fidèle* des sous-domaines $\{x : t \mid p(x)\}$, c'est-à-dire de construire un type dont les valeurs correspondraient exactement aux objets v de type t qui satisfont le prédicat $p(v)$.
  C'est à l'utilisateur de justifier que les valeurs manipulées vérifient le prédicat $p(x)$ par exemple en effectuant une opération de test lors de la construction d'un objet.
- CAML nous permet de distinguer plusieurs types partageant la même représentation interne.
- Le mécanisme de filtrage permet d'accéder aux composantes d'un objet structuré.

4.3 Domaine fini

4.3.1 Spécification

On suppose que l'on veut modéliser un domaine comportant exactement n valeurs.

Ce domaine est spécifié par un ensemble fini de constantes $v_1, \ldots v_n$ telles que $v_i \neq v_j$ lorsque $i \neq j$ et telle que toute valeur dans le domaine est égale à l'un des v_i.

Transparent

Forme générale La forme générale de la spécification d'un domaine fini est :

SPÉCIFICATION:	
{Domaine}	
dom	{Domaine fini représentant ...}
{Constantes}	
$v_1, \ldots, v_n$:*dom*	{Les n constantes distinctes du domaine *dom*}

Fin 4.13

4.3.2 Exemples

Transparent

Exemple 1 Les jours de la semaine :

SPÉCIFICATION:	
{Domaine}	
`jour`	{Représente l'ensemble fini des jours de la semaine}
{Constantes}	
`lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche : jour`	
{les 7 valeurs distinctes du domaine `jour`}	

Fin 4.14

Transparent

Exemple 2 Les couleurs d'un jeu de cartes :

SPÉCIFICATION:	
{Domaine}	
`couleur`	{Représente l'ensemble fini des couleurs d'un jeu de cartes}
{Constantes}	
`pique, coeur, carreau, trèfle : couleur`	
{les 4 valeurs distinctes du domaine couleur}	

Fin 4.15

Transparent

Exemple 3 Les figures d'un jeu de belote :

SPÉCIFICATION:	
{Domaine}	
`figure`	{Représente l'ensemble fini des figures d'un jeu de belote}
{Constantes}	
`as, roi, dame, valet, dix, neuf, huit, sept : figure`	
{les 8 valeurs distinctes du domaine figure}	

Fin 4.16

4.3.3 Réalisation

On pourrait réaliser un tel domaine en utilisant une restriction du type primitif des chaînes de caractères ou des entiers. Dans les deux cas, une difficulté est que les fonctions manipulant ces objets doivent aussi considérer les cas dégénérés où l'entrée est une chaîne ou un entier ne correspondant pas à l'un des objets du domaine fini que l'on modélise. De plus, si on sait qu'une donnée ne peut prendre qu'un petit nombre de valeurs, on peut la représenter en machine de manière efficace (un entier est codé sur 32 bits, alors que 4 bits sont suffisants pour représenter un ensemble à 16 éléments). Le test d'égalité entre deux chaînes de caractères a également un coût proportionnel à la longueur de la chaîne, ce qui pénalise l'utilisation de noms longs qui sont pourtant essentiels à la lisibilité du programme.

Tout cela suggère une représentation directe plus efficace des ensembles finis. Le langage CAML propose une telle construction.

4.3.4 Codage en CAML

En CAML, il est possible d'introduire un type dont on énumère les valeurs :

Transparent

```
type couleur = Pique | Coeur | Carreau | Trefle;;
type figure = As | Roi | Dame | Valet | Dix | Neuf | Huit | Sept;;
```

ou plus généralement :

$$\texttt{type}\ \mathit{nouveau\_type} = \mathit{Const}_1 \mid \ldots \mid \mathit{Const}_n$$

Fin 4.17

L'utilisateur choisit le nom du type et des constantes. Les constantes ainsi introduites sont distinctes et forment les seules valeurs possibles du type, en particulier une constante est un motif correct qui peut servir dans un filtrage.

Ainsi on pourra écrire :

Transparent

```
#Pique=Coeur;;
- : bool = false
#let est_rouge x = match x with
    Pique   -> false
  | Coeur   -> true
  | Carreau -> true
  | Trefle  -> false;;
val est_rouge : couleur -> bool = <fun>
#est_rouge Carreau ;;
- : bool = true
```

Fin 4.18

La représentation en CAML des domaines finis est *fidèle* puisque les valeurs de ce type correspondent exactement aux valeurs du domaine.

4.4 Domaine Produit

4.4.1 Spécification

Soient $d_1, \ldots, d_n$ des domaines déjà construits. Soit à modéliser un domaine *dom* dont les valeurs peuvent se ramener à des n-uplets $(v_1, \ldots, v_n)$ avec $v_i \in d_i$. On dit que le domaine *dom* est le *produit* des domaines $d_1, \ldots, d_n$. L'exemple des points introduit au début de ce cours est un domaine produit de deux entiers.

Les opérations essentielles pour manipuler un objet dans ces domaines produit sont la formation d'un nouvel objet à partir de ses composantes et l'accès à l'une des composantes de l'objet.

Le domaine produit de $d_1 \ldots d_n$ se spécifie de la manière suivante :

Transparent

SPÉCIFICATION:

{Domaine}
dom {Produit de $d_1 \ldots d_n$}
{Fonction de construction}
$cons\_dom\ (x_1 : d_1; \ldots ; x_n : d_n) \rightarrow dom$
{Construit un n-uplet dans *dom*}
{Fonctions d'accès}
$acces_1\ (p : dom) \rightarrow d_1$ {Accès à la première composante de p}
$\vdots$
$acces_n\ (p : dom) \rightarrow d_n$ {Accès à la n-ième composante de p}

Fin 4.19

Les propriétés de cette spécification sont que :

$$acces_i(cons\_dom(e_1, \ldots, e_n)) = e_i$$

on en déduit que :

$$cons\_dom(e_1, \ldots, e_n) = cons\_dom(f_1, \ldots, f_n) \Leftrightarrow \forall 1 \leq i \leq n.e_i = f_i$$

Transparent

Exemple 1 Une carte est formée d'une figure et d'une couleur.

SPÉCIFICATION:

{Domaine}
`carte` {représente une carte par une figure et une couleur}
{Fonction de construction}
`cons_carte` (f : `figure`; c : `couleur`) $\rightarrow$ `carte`
{Fonctions d'accès}
`figure_de_carte` (c: `carte`) $\rightarrow$`figure`
`couleur_de_carte` (c: `carte`) $\rightarrow$`couleur`

Fin 4.20

Transparent

Exemple 2 Une main de Poker est formée de 5 cartes.

SPÉCIFICATION:
{Domaine}
`main` {représente une main de poker composée de 5 cartes}
{Fonction de construction}
`cons_main` (c_1, c_2, c_3, c_4, c_5 : `carte`) $\rightarrow$ `main`
{Fonctions d'accès}
`carte1,carte2,carte3,carte4,carte5` (m: `main`) $\rightarrow$`carte`

Fin 4.21

Transparent

Exemple 3 Un abonné est représenté par un nom et un numéro d'abonné :

SPÉCIFICATION:
{Domaine}
`abonné` {repésente un abonné par son nom et son numéro d'abonné}
{Fonction de construction}
`cons_abonné` (n : `nom`; m : `numéro`) $\rightarrow$ `abonné`
{Fonctions d'accès}
`nom_abonné` (a : `abonné`) $\rightarrow$ `nom`
`numéro_abonné` (a : `abonné`) $\rightarrow$ `numéro`

Fin 4.22

Notations pour un produit générique Dans de nombreuses modélisations, introduire explicitement un nouveau nom pour un domaine représentant un produit et les fonctions de construction et d'accès aide à la compréhension de la spécification. Il arrive que ce ne soit pas le cas mais que l'on ait besoin d'un domaine produit comme résultat d'une opération pour calculer de manière efficace simultanément deux objets. C'est le cas par exemple si on veut, étant donné deux nombres, calculer à la fois le minimum et le maximum en ne faisant qu'un seul test.

Pour ces cas, on propose d'utiliser une notation pour un produit générique.

Transparent

Notation pour un produit générique

Si $d_1, \ldots, d_n$ sont des domaines alors on notera $d_1 \times \ldots \times d_n$ le domaine correspondant au produit des d_i. Le constructeur de ce domaine est noté `n-uple` mais on écrira plus simplement $(e_1, \ldots, e_n)$ pour `n-uple`$(e_1, \ldots, e_n)$ la projection sur la i-ème composante de profil $d_1 \times \ldots \times d_n \rightarrow d_i$ sera notée $proj_i$.

On pourra donc écrire :

```
min_max : (x,y: entier) → entier × entier
{renvoie la paire ordonnée des deux entiers}
min_max : (x,y) ↦ si x < y alors:(x,y) sinon: (y,x)
```

Fin 4.23

Transparent

On introduit la nouvelle expression :

$$\texttt{soit}\ (x_1, \ldots, x_n) = \textit{expr},\ \textit{expr'}$$

comme abréviation de

$$\texttt{soit}\ x_1 = \textit{proj}_1(\textit{expr}),\ \ldots,\ x_n = \textit{proj}_n(\textit{expr}),\ \textit{expr'}$$

La fonction `min` pourra alors s'écrire :

```
min : (x,y) ↦ soit (m1,m2)= min_max(x,y), m1
```

Fin 4.24

4.4.2 Codage dans CAML

CAML offre trois manières distinctes de coder un produit de plusieurs types : le produit générique, le produit étiqueté et les enregistrements.

Produit générique

CAML, contrairement à de nombreux langages de programmation, offre une construction de types produit générique analogue à celle que nous avons introduite dans notre langage de description.

Transparent

Si $t_1, \ldots, t_n$ sont des types alors $t_1 * \ldots * t_n$ est un type.

Si a_1 a pour type t_1,...et a_n a pour type t_n alors le n-uple $(a_1, \ldots, a_n)$ a pour type $t_1 * \ldots * t_n$.

Si e et f sont des expressions et $x_1, \ldots, x_n$ sont des variables alors on peut écrire l'expression CAML `let` $(x_1, \ldots, x_n)$ $= e$ `in` f.

Fin 4.25

Dans cette expression, chaque variable x_i est une nouvelle variable de type t_i dont la portée est réduite à l'expression f.
L'expression `let` est bien formée de type t si e a pour type $t_1 * \ldots * t_n$ et f a pour type t.
Lorsque e s'évalue en le n-uplet $(v_1, \ldots, v_n)$ alors l'expression `let`... s'évalue en

$$\texttt{let}\ x_1 = v_1\ \texttt{and}\ \ldots\ \texttt{and}\ x_n = v_n\ \texttt{in}\ f$$

On peut écrire l'exemple du minimum en CAML de la façon suivante :

Transparent

```
let min_max x y  = if x < y then (x,y) else (y,x);;
let min x y  = let (m1,m2)=(min_max x y) in m1;;
```

Remarque: une fonction peut prendre un couple comme argument unique.

```
let min_max2 (x,y) = if x < y then (x,y) else (y,x);;
let min x y  = let (m1,m2)=min_max2(x,y) in m1;;
```

La définition de `min_max2` est équivalente à

```
let min_max2 z = let (x,y) = z in if x < y then (x,y) else (y,x);;
```

Fin 4.26

Dans l'expression `let` $(x_1, \ldots, x_n)$ $= e$ `in` f, si certaines variables x_i n'apparaissent pas dans f alors on peut les remplacer par le caractère spécial `_`.

Type étiqueté

La formation d'un type produit ci-dessus n'introduit pas de nouveau type. On peut remédier à ce problème en utilisant les types étiquetés associés à des produits :

Transparent

$$\texttt{type}\ \mathit{nouveau\_type}\ = \mathit{Etiquette}\ \texttt{of}\ t_1 * \ldots * t_n$$

$$\texttt{let}\ \mathit{cons\_dom}\, x_1 \ldots x_n = \mathit{Etiquette}(x_1, \ldots, x_n)$$

$$\texttt{let}\ \mathit{acces}_i(e) = \texttt{match}\ e\ \texttt{with}\ \mathit{Etiquette}(x_1, \ldots, x_n) \rightarrow x_i$$

Fin 4.27

Transparent

Exemple 1-Codage CAML

```
#type carte = Carte of figure * couleur;;
#let cons_carte f c  = Carte(f,c);;
val cons_carte : figure -> couleur -> carte = <fun>
#let figure_de_carte e  = match e with Carte(f,c)->f;;
val figure_de_carte : carte -> figure = <fun>
#let couleur_de_carte e = match e with Carte(f,c)->c;;
val couleur_de_carte : carte -> couleur = <fun>
```

Fin 4.28

Types enregistrements

Les produits étiquetés ne sont pas satisfaisants lorsque l'on manipule des produits dont plusieurs composantes ont le même type, c'est le cas par exemple d'une *date* qui est représentée par trois entiers pour le jour, le mois et l'année. On risque alors de mélanger les composantes.

CAML offre une troisième construction pour représenter des produits qui est la construction de type enregistrement. Un intérêt de cette construction est qu'on la retrouve également dans les langages de programmation impératifs. L'idée du type enregistrement est d'associer des étiquettes à chacune des composantes du produit.

Transparent

Si $t_1, \ldots, t_n$ sont des types alors on peut introduire un nouveau type enregistrement :

$$\texttt{type}\ \mathit{nouveau\_type}\ = \{\mathit{etiq}_1 : t_1; \ldots; \mathit{etiq}_n : t_n\}$$

Si a_1 a pour type t_1, …, et a_n a pour type t_n alors $\{\mathit{etiq}_1 = a_1; \ldots; \mathit{etiq}_n = a_n\}$ a pour type *nouveau_type*.

Si e a pour type *nouveau_type* alors $e.\mathit{etiq}_i$ a pour type t_i.

On a : $\{\mathit{etiq}_1 = a_1; \ldots; \mathit{etiq}_n = a_n\}.\mathit{etiq}_i = a_i$

Fin 4.29

Ces opérations de base permettent le codage des fonctions de construction et d'accès d'un domaine produit :

Transparent

$$\texttt{type}\ \mathit{nouveau\_type}\ = \{etiq_1 : t_1; \ldots; etiq_n : t_n\}$$

$$\texttt{let}\ \mathit{cons\_dom}\, x_1\ \ldots\ x_n = \{etiq_1 = x_1; \ldots; etiq_n = x_n\}$$

$$\texttt{let}\ acces_i(e) = e.etiq_i$$

Fin 4.30

Transparent

Exemple

```
#type carte = {fig:figure; col:couleur};;
#let cons_carte f c  = {fig=f;col=c};;
val cons_carte : figure -> couleur -> carte = <fun>
#let figure_de_carte  e  = e.fig;;
val figure_de_carte : carte -> figure = <fun>
#let couleur_de_carte  e = e.col;;
val couleur_de_carte : carte -> couleur = <fun>
```

Fin 4.31

Il est également possible d'utiliser des motifs pour accéder aux composantes d'un objet enregistrement. Un motif de type enregistrement est de la forme $\{etiq_1 = x_1; \ldots; etiq_n = x_n\}$ avec x_i des variable de type t_i dont la portée sera restreinte au membre droit f du filtrage. Si certains x_i ne sont pas utilisés dans f alors il n'est pas nécéssaire de mentionner l'étiquette correspondante.

La fonction d'accès peut donc s'écrire :

$$\texttt{let}\ acces_i(e) = \texttt{match}\ e\ \texttt{with}\ \{etiq_i = x\} \rightarrow x$$

Transparent

Exemple

```
#let figure_de_carte e = match e with {fig=f} -> f;;
val figure_de_carte : carte -> figure = <fun>
#let couleur_de_carte e = match e with {col=c} -> c;;
val couleur_de_carte : carte -> couleur = <fun>
```

Fin 4.32

4.5 Somme

4.5.1 Spécification

Transparent

Supposons qu'on veuille modéliser un domaine *dom* qui est composé d'objets appartenant soit à un domaine d_1 soit à un domaine d_2 et pour lesquels on peut dire auquel des deux domaines l'objet appartient (on parle de *somme disjointe* qui est différente de l'union ensembliste). On dit que le domaine *dom* est la somme des domaines d_1 et d_2.

Fin 4.33

Introduction

Par exemple un exemplaire de la bibliothèque peut être soit un livre soit une vidéo. Soit `video` le domaine pour représenter les vidéos et `livre` le domaine pour représenter les livres que nous supposerons déjà construits. Le domaine `exemplaire` pour représenter les exemplaires doit comporter deux opérations, permettant de passer respectivement d'un objet du domaine `livre` à un objet du domaine `exemplaire` et d'un objet du domaine `video` à un objet du domaine `exemplaire`, ce sont les *opérations de construction*. Ensuite étant donné un objet dans le domaine `exemplaire`, une *opération de test* permet de déterminer si l'exemplaire est un livre ou une vidéo. Finalement deux *opérations d'accès* permettent de repasser d'un objet dans le domaine `exemplaire` à un objet dans le domaine `video` (respectivement dans le domaine `livre`).

On parle de somme disjointe car on peut être amené à faire la somme de deux fois le même domaine. On pourra par exemple modéliser des sommes d'argent qui sont soit exprimées en francs, soit exprimées en euros. Étant donné une somme d'argent, il faut pouvoir déterminer si c'est une somme en francs ou en euros, il faut donc garder une trace dans la représentation de la somme d'argent, de comment elle a été construite.

Cas général

On décrit maintenant la spécification générale de la somme disjointe de n domaines $d_1, \ldots, d_n$. Soit *dom* le domaine des valeurs de la somme disjointe de $d_1, \ldots, d_n$. Toute valeur de d_i permet de construire une valeur de *dom*. Étant donnée une valeur v de *dom* on aimerait récupérer la composante dans d_1 ou ... ou d_n suivant comment est construite v. Cet accès se décompose en une opération de test qui permet de savoir si la valeur v est construite à partir d'un élément de d_1 ou ... ou d_n puis en n opérations d'accès distinctes suivant si on est dans d_1 ou ... ou d_n.

Transparent

SPÉCIFICATION:

{Domaine}
dom {Somme disjointe de n domaines $d_1, \ldots, d_n$}
{Fonctions de construction}
$cons_1\ (x : d_1) \rightarrow dom$ {Premier constructeur}
...
$cons_n\ (x : d_n) \rightarrow dom$ {n-ème constructeur}
{Fonctions de test}
$est_1, \ldots, est_n : dom \rightarrow$`booléen`
{Fonctions d'accès}
val_1: $(x : dom)\{est_1(x)\} \rightarrow d_1$
...
val_n: $(x : dom)\{est_n(x)\} \rightarrow d_n$

Fin 4.34

Les propriétés de ces opérations sont : $est_i(cons_i(v)) =$ `vrai` et $est_i(cons_j(v)) =$ `faux` si $i \neq j$ ainsi que $val_i(cons_i(v)) = v$

On remarque que les fonctions d'accès ne sont définies que partiellement sur des sous-domaines de *dom*. On pourrait ne spécifier de manière abstraite que $n-1$ fonctions de tests, la n-ème pouvant s'exprimer comme la négation des $n-1$ autres. Il est souvent plus agréable, pour des raisons d'uniformité, d'introduire une opération de test par constructeur.

Transparent

Exemple 1 : bibliothèque Un exemplaire à la bibliothèque est soit un livre soit une video.

SPÉCIFICATION:
{Domaine}
`exemplaire` {Un exemplaire est soit un livre soit une video}
{Fonctions de construction}
`cons_livre` (l:`livre`) → `exemplaire`
`cons_video` (v:`video`) → `exemplaire`
{Fonctions de test}
`est_livre`, `est_video` (e: `exemplaire`) → `booléen`
{Fonctions d'accès}
`ex_livre` : (e:`exemplaire`){`est_livre`(e)} → `livre`
`ex_video` : (e:`exemplaire`){`est_video`(e)} → `video`

Fin 4.35

Transparent

Exemple 2 : somme en francs ou en euros Une somme d'argent est comptée soit en francs soit en euros.

SPÉCIFICATION:
{Domaine}
`argent` {somme d'argent comptée soit en francs soit en euros}
{Fonctions de construction}
`francs`(n:`entier`) → `argent`
`euros`(n:`entier`) → `argent`
{Fonctions de test}
`est_francs`, `est_euros`(x:`argent`) → `booléen`
{Fonctions d'accès}
`val_francs` (e:`argent`){`est_francs`(e)} → `entier`
`val_euros` (e:`argent`){`est_euros`(e)} → `entier`

Fin 4.36

Transparent

Exemple 3 : cartes de tarots On suppose donné un type `carte` qui modélise les cartes ordinaires du tarot par une figure et une couleur.

Une carte de tarot est soit une figure et une couleur soit un atout représenté par un numéro entre 0 et 21. La contrainte sur le nombre est ajouté au constructeur d'atout. On peut introduire également un constructeur étendu qui teste la contrainte.

SPÉCIFICATION:	
{Domaine}	
`carte_tarot`	{Une carte de tarot est soit une carte ordinaire soit un atout}
{Fonctions de construction}	
`cons_carte` (c:`carte`) $\rightarrow$ `carte_tarot`	
`cons_atout` (n:`entier`)$\{0 \leq n \leq 21\}$ $\rightarrow$ `carte_tarot`	
{l'entier 0 servira à représenter l'excuse}	
{Fonctions de construction étendues}	
`entier_to_atout`(n: entier) $\rightarrow$ `carte_tarot`	
{Fonctions de test}	
`est_atout, est_carte (c:carte_tarot)` $\rightarrow$ `booléen`	
{Fonctions d'accès}	
`carte_de_tarot (x:carte_tarot){est_carte(x)}` $\rightarrow$ `carte`	
`atout_de_tarot`(x:`carte_tarot`){`est_atout`(x)} $\rightarrow$ `entier`	

Fin 4.37

4.5.2 Codage CAML

Le codage CAML se fait à l'aide d'un type somme qui associe une étiquette à chaque alternative possible dans la construction des objets.

Transparent

$$\texttt{type}\ \mathit{nouveau\_type} = \mathit{Constr}_1\ \texttt{of}\ t_1 |\ \ldots\ |\ \mathit{Constr}_n\ \texttt{of}\ t_n$$

Si v est une valeur de type t_i alors $\mathit{Constr}_i(v)$ est une valeur de type *nouveau_type*.

Une expression définie par filtrage sur un programme e dans un type somme devra considérer tous les cas possibles pour la valeur e.

$$\texttt{match}\ e\ \texttt{with}\ \mathit{Constr}_1(x_1) \rightarrow f_1\ |\ \ldots\ |\ \mathit{Constr}_n(x_n) \rightarrow f_n$$

Dans cette expression chaque variable x_i a pour type t_i et sa portée est limitée à f_i. Lorsque e s'évalue en $\mathit{Constr}_i(v_i)$, la variable x_i est liée à la valeur v_i.

Fin 4.38

On peut coder les opérations du domaine somme à l'aide du type somme. Les étiquettes servent pour les fonctions de construction ; le filtrage permet de définir les fonctions de test et d'accès pour les types sommes :

Transparent

$$\texttt{type}\ \mathit{nouveau\_type} = \mathit{Constr}_1\ \texttt{of}\ t_1 |\ \ldots\ |\ \mathit{Constr}_n\ \texttt{of}\ t_n$$

$$\texttt{let}\ \mathit{cons}_i\ d = \mathit{Constr}_i(d)$$

$$\texttt{let}\ \mathit{est}_i\ e = \texttt{match}\ e\ \texttt{with}\ \mathit{Constr}_i(d) \rightarrow \texttt{true}\ |\ \_ \rightarrow \texttt{false}$$

$$\begin{array}{llll} \texttt{let}\ \mathit{val}_i\ e = & \texttt{match}\ e & \texttt{with}\ \mathit{Constr}_i(d) & \rightarrow d \\ & & |\ \_ & \rightarrow \texttt{failwith "constructeur erroné"} \end{array}$$

Fin 4.39

Exemples

Transparent

```
#type argent = Francs of int | Euros of int;;
#let francs f =Francs f;;
val francs : int -> argent = <fun>
#let euros f =Euros f;;
val euros : int -> argent = <fun>
#let est_francs s = match s with Francs(x) -> true | _ -> false;;
val est_francs : argent -> bool = <fun>
#let est_euros s = match s with Euros(x) -> true | _ -> false;;
val est_euros : argent -> bool = <fun>
#let val_francs s =
    match s with Francs(x) -> x | _ -> failwith "Somme en euros";;
val val_francs : argent -> int = <fun>
#let val_euros s =
    match s with Euros(x) -> x | _ -> failwith "Somme en francs";;
val_euros : argent -> int = <fun>
```

Fin 4.40

Transparent

```
#type carte_tarot = Cartet of carte | Atout of int;;
#let cons_carte c = Cartet c;;
val cons_carte : carte -> carte_tarot = <fun>
#let cons_atout n = Atout n;;
val cons_atout : int -> carte_tarot = <fun>
#let int_to_atout n = if 0 <= n & n <= 21
                                 then Atout n
                                 else failwith"numéro d'atout invalide";;
val int_to_atout : int -> carte_tarot = <fun>
#let est_carte c = match c with Cartet(_) -> true | _ -> false;;
#let est_atout c = match c with Atout(_) -> true | _ -> false;;

#let carte_de_tarot c = match c with Cartet(x) -> x
                                     | _ -> failwith "Pas une carte";;

#let atout_de_tarot c = match c with Atout(x) -> x
                                     | _ -> failwith "Pas une carte";;
```

Fin 4.41

On remarque dans cet exemple que la fonction de construction `cons_atout` n'est pas codée comme une simple application du constructeur du type CAML `Atout` mais effectue un test sur son argument afin de vérifier qu'il appartient bien au sous-domaine spécifié.

Transparent

Type des expressions CAML

```
val cons_carte : carte -> carte_tarot = <fun>
val cons_atout : int -> carte_tarot = <fun>
val est_carte : carte_tarot -> bool = <fun>
val est_atout : carte_tarot -> bool = <fun>
val carte_de_tarot : carte_tarot -> carte = <fun>
val atout_de_tarot : carte_tarot -> int = <fun>
```

Fin 4.42

4.6 Combinaison constantes, sommes et produits

En général, un domaine réel se modélise comme une union de différents sous-domaines parfois réduits à un point, parfois composés comme produit de domaines plus simples. Les représenter à l'aide des constructions vues précédemment est possible mais fait intervenir un grand nombre de types abstraits ce qui alourdit l'écriture sans toujours clarifier la modélisation.

La spécification du type abstrait peut se faire directement de la manière suivante :
- nommer le type
- introduire une constante par objet individuel dans le domaine
- introduire un constructeur par sous-domaine, les arguments de ce constructeur correspondant à la décomposition du sous-domaine en produits
- une fonction de test par constructeur lorsqu'il y a plus d'un constructeur
- une fonction d'accès par domaine apparaissant comme argument dans le profil d'un constructeur

4.6.1 Exemples

Exemple 1 : Cartes de tarots On peut reprendre notre modélisation des cartes de tarots, en choisissant un modélisation moins ad-hoc de l'excuse et sans passer par le type intermédiaire des cartes :

Transparent

SPÉCIFICATION:
{Domaine}
`carte_tarot`
{Constantes}
`excuse : carte_tarot`
{Fonctions de construction}
`cons_carte (f:figure;c:couleur)` $\rightarrow$ `carte_tarot`
`cons_atout` (n:`entier`){$1 \leq n \leq 21$} $\rightarrow$`carte_tarot`
{Fonctions de construction étendues}
`entier_to_atout`(n:`entier`) $\rightarrow$`carte_tarot`
{Fonctions de test}
`est_atout, est_carte (c:carte_tarot)` $\rightarrow$ `booléen`
{Fonctions d'accès}
`figure_de_carte_tarot` : (c:`carte_tarot`){`est_carte`(c)} $\rightarrow$`figure`
`couleur_de_carte_tarot` : (c:`carte_tarot`){`est_carte`(c)} $\rightarrow$`couleur`
`atout_de_carte_tarot` (c:`carte_tarot`){`est_atout`(c)} $\rightarrow$ `entier`

Fin 4.43

On remarque que le constructeur `cons_carte` attend deux arguments (une figure et une couleur) alors que le constructeur `cons_atout` ne prend qu'un argument. Les valeurs de ce domaine sont soit la constante `excuse`, soit de la forme `cons_carte`(f, c) avec f une figure et c une couleur, soit de la forme `cons_atout`(n) avec n un entier compris entre 1 et 21. On pourrait introduire une fonction de test `est_excuse` mais nous utiliserons plus directement le test $x =$ `excuse` à la place de `est_excuse`(x)

Exemple 2 : nombre variable de données Certaines opérations doivent renvoyer un nombre variable de données. Par exemple une équation du second degré peut avoir 0, 1 ou 2 racines réelles. Une opération qui calcule ces racines doit pouvoir renvoyer $0, 1$ ou 2 valeurs.

On modélise aisément un domaine `solution` permettant de repésenter le résultat de cette fonction.

Transparent

SPÉCIFICATION:
{Domaine}
`solution`
{Constantes}
`pas_de_racine` : `solution`
{Fonctions de construction}
`une_racine` (r:`réel`) → `solution`
`deux_racines` (r,r':`réel`) →`solution`
{Fonctions de test}
`a_une_racine`,`a_deux_racines` (s:`solution`) → `booléen`
{Fonctions d'accès}
`racine` (s:`solution`){`a_une_racine`(s)} → `réel`
`racine1`,`racine2` (s:`solution`){`a_deux_racines`(s)} → `réel`

Fin 4.44

On peut alors spécifier l'opération de résolution d'équations du second degré et la réaliser.

Transparent

SPÉCIFICATION:
`resoud_equation` (a:`réel`*;b,c:`réel`) → `solution`
{retourne les solutions de l'équation $ax^2 + bx + c = 0$}

RÉALISATION:
`resoud_equation` : (a,b,c) ↦
soit `delta` = b*b-4*a*c,
selon le cas `delta`
`delta` < 0: `pas_de_racine`
`delta` = 0: `une_racine`$(-b/(2*a))$
`delta` > 0: **soit** d = `racine_carree(delta)`, `deux_racines`$((-b+d)/(2*a), (-b-d)/(2*a))$

Fin 4.45

4.6.2 Codage CAML

Le codage en CAML est une généralisation de ce que nous avons vu précédemment pour les types finis, produits ou sommes. On introduit un nouveau type dans lequel on fait apparaître les constantes et les constructeurs, ces derniers étant associés aux types de leurs arguments.

Transparent

```
#type carte_tarot = Excuse | Carte of figure * couleur | Atout of int;;

#let excuse = Excuse;;
val excuse : carte_tarot = Excuse

#let figure f c=Carte(f,c);;
val figure : figure -> couleur -> carte_tarot = <fun>
#let atout n = Atout(n);;
val atout : int -> carte_tarot = <fun>
#let int_to_atout n = if 1<=n & n <= 21 then Atout(n)
               else failwith"numéro d'atout invalide";;
val int_to_atout : int -> carte_tarot = <fun>
```

Fin 4.46

Transparent

```
#let est_carte c = match c with Carte(f,d) -> true | _ -> false;;
#let est_atout c = match c with Atout(x) -> true | _ -> false;;

#let figure_de_carte_de_tarot c =
       match c with Carte(f,d) -> f | _ -> failwith "Pas une carte";;
val figure_de_carte_de_tarot : carte_tarot -> figure = <fun>

#let couleur_de_carte_de_tarot c =
       match c with Carte(f,d) -> d | _ -> failwith "Pas une carte";;
val couleur_de_carte_de_tarot : carte_tarot -> couleur = <fun>

#let atout_de_tarot c = match c with Atout(x) -> x
                                    | _ -> failwith "Pas une carte";;
val atout_de_tarot : carte_tarot -> int = <fun>
```

Fin 4.47

4.6.3 Filtrage général

L'opération de définition de filtrage est très générale. Elle sert à la fois à tester la forme d'une expression et à lier des variables à des sous-expressions. Dans le filtrage, apparaissent des motifs qui sont composés :

- soit d'une constante dans un type de base ou défini par l'utilisateur;
- soit du symbole spécial `_`;
- soit d'une variable;
- soit d'un constructeur `C` d'un type défini par l'utilisateur appliqué à p autres motifs, si le constructeur attend p arguments, le motif s'écrit alors $\mathtt{C}(m_1, \ldots, m_p)$;
- soit d'un motif enregistrement.

Transparent

Ainsi une fonction comptant les points associés aux cartes aux tarots pourra se coder de la manière suivante :

```
let points c = match c with
    Excuse        -> 4.5
  | Carte(f,_)    -> (match f with Roi      -> 4.5
```

```
                                        | Dame     -> 3.5
                                        | Cavalier -> 2.5
                                        | Valet    -> 1.5
                                        | _        -> 0.5)
  | Atout(1)      -> 4.5
  | Atout(21)     -> 4.5
  | _             -> 0.5;;
val points : carte_tarot -> float = <fun>
```

Fin 4.48

On remarque les parenthèses autour de l'expression de filtrage imbriquée, en l'absence de ces parenthèses la suite des motifs pour le filtrage de `c` serait interprétée comme la suite des motifs du filtrage de `f`.

Transparent

Toute fonction définie par filtrage peut être traduite en une expression n'utilisant que les fonctions de test et d'accès.

```
let points c =
    if c = excuse then 4.5
    else if (est_carte c) then let f = (figure_de_carte_de_tarot c) in
         if f = Roi then 4.5 else if f = Dame then 3.5 else
         if f = Cavalier then 2.5 else if f = Valet then 1.5 else 0.5
    else let n = (atout_de_carte_de_tarot c) in
         if n=1 or n=21 then 4.5 else 0.5;;
```

Fin 4.49

Pièges à éviter

- Lorsque l'on imbrique des définitions par filtrage, il faut faire attention à parenthéser les filtrages internes, une alternative peut être de définir des fonctions auxiliaires.
- Une déclaration de type CAML introduit un nouveau nom pour un type et des étiquettes. Ces étiquettes restent associées au type déclaré dans toute la suite du programme sauf si un nouveau type est introduit qui réutilise le même nom d'étiquette (même associée à un autre type). Une fois cette redéclaration faite, l'étiquette fera toujours référence au second type, l'étiquette du premier type n'est plus accessible. Il est donc vivement déconseillé d'utiliser le même nom d'étiquette dans plusieurs déclarations distinctes.

4.7 Types abstraits généraux

4.7.1 Opérations structurelles et autres fonctions

Nous avons vu précédemment comment structurer des spécifications pour représenter des domaines particuliers. Lorsqu'on modélise un problème, on est amené à spécifier le domaine d'étude, de manière abstraite lorsqu'il s'agit de données structurées puis à spécifier et éventuellement réaliser des fonctions spécifiques au problème abordé.

Les fonctions se divisent donc en deux classes : les fonctions servant à modéliser la structure du domaine et qui seront directement codées dans le langage de programmation (il s'agit des constantes, des constructeurs, opérations de tests et d'accès), qu'on appelle les *opérations structurelles* ; puis des fonctions liées au problème abordé qui pourront être réalisées à partir des opérations structurelles, que nous appelerons les *autres fonctions*.

Exemple Les nombres rationnels peuvent se modéliser de la manière suivante :

Transparent

SPÉCIFICATION:	
{Domaine}	
`rat`	{Le domaine des nombres rationnels}
{Fonction de construction}	
`cons_rat` (p:`entier`;q:`entier`$^*_+$) →`rat`	{Le rationnel $\frac{p}{q}$ avec $q > 0$}
{Fonction de construction étendue}	
`entiers_to_rat` (p,q:`entier`) →`rat`	{Construit un rationnel équivalent à $\frac{p}{q}$}
{Fonctions d'accès }	
`num`(r:`rat`)→`entier`	{calcule le numérateur de r}
`den`(r:`rat`)→`entier`$^*_+$	{calcule le dénominateur de r}
{Autres fonctions}	
`rplus` (r1,r2: `rat`) → `rat`	{calcule $r1 + r2$}
`rmult` (r1,r2: `rat`) → `rat`	{calcule $r1 * r2$}

Fin 4.50

Transparent

RÉALISATION:

`rplus` : (r,r') ↦ `soit` d = `den`(r) `,` d' = `den`(r') `,` n = `num`(r) `,` n' = `num`(r'),
`cons_rat(n*d'+n'*d,d*d')`

`rmult` : (r,r') ↦ `cons_rat(num(r)* num(r'),den(r)*den(r'))`

Fin 4.51

Avantage des types abstraits. On peut modifier facilement une petite partie d'un gros programme. Par exemple, le type abstrait “dictionnaire d'entiers” permet de représenter un ensemble d'entiers, d'y insérer ou d'en supprimer un élément, et de rechercher si un entier x lui appartient. Dans l'état actuel de vos connaissances, si on vous demande de coder un dictionnaire d'entiers, vous utiliserez une séquence d'entiers. A la fin de ce cours, vous pourriez envisager de remplacer ce codage par un codage avec des arbres, beaucoup plus efficace. Si à l'usage vous découvrez que vous manquez de place mémoire, vous pourriez alors envisager de remplacer ce codage par un codage dans un tableau, plus compact. Dans tous les cas, vous n'aurez à modifier qu'un petit morceau de programme, et le reste du programme est inchangé dans la mesure où les dictionnaires sont manipulés uniquement par l'intermédiaire des opérations spécifiées dans la définition du type abstrait : le reste du programme fait comme s'il ne connaissait pas les détails du codage d'un dictionnaire.

Ainsi, on peut réfléchir à la partie du programme “codage du type abstrait dictionnaire” et au reste du programme de manière entièrement séparée, ces deux parties ne communiquent que via les spécifications. Les deux parties du programme peuvent être confiées à des programmeurs différents : si ceux-ci respectent les spécifications avec exactitude, leurs morceaux de programmes seront compatibles.

Transparent

4.7.2 Principe de la modélisation des données

Les données sont modélisées par un domaine, on n'oubliera pas de :
- donner un nom au domaine;
- spécifier les fonctions définies sur les objets du domaine en indiquant leur profil;
- identifier les opérations structurelles :
  - valeurs constantes,
  - constructeurs,
  - fonctions de tests de constructeurs,
  - opérations d'accès aux composantes;
- mettre des commentaires.

Fin 4.52

4.7.3 Codage implicite/explicite

Dans une spécification, on introduit des domaines abstraits dont les opérations structurelles (constructeurs, fonctions de tests et d'accès) sont spécifiées sans être réalisées. Les autres fonctions sur le domaine peuvent être réalisées à l'aide des opérations structurelles et des opérations sur les sous-domaines. C'est le cas de l'exemple précédent des entiers rationnels. Lorsqu'on effectue le codage du domaine, un choix est fait sur le type qui servira à représenter les valeurs du domaine, et les opérations structurelles vont être codées en utilisant les primitives du langage de programmation (en particulier le filtrage dans le cas de CAML).

Transparent

Lorsqu'on effectue le codage des autres fonctions alors deux choix sont possibles :
- Le codage explicite ou abstrait : on ne fait appel qu'aux opérations structurelles apparaissant dans la spécification.
- Le codage implicite ou concret : on utilise les primitives du langage de programmation associées à la représentation du domaine pour coder les autres opérations.

Fin 4.53

Dans le premier cas (abstrait) le codage des autres fonctions ne dépend pas du choix de représentation pour le domaine. Ainsi le changement de représentation ne nécessite que de recoder les opérations structurelles. Le codage des autres fonctions est également une simple traduction de leur réalisation. Un codage abstrait est également souvent moins sensible qu'un codage concret à un changement dans la modélisation : par exemple, si on décide d'ajouter une composante "couleur" à un point, les opérations structurelles `abscisse` ou `ordonnée` conservent le même profil et le même sens.

À l'opposé, le codage implicite suppose la connaissance de la représentation du type, ce qui permet d'utiliser les mécanismes comme le filtrage qui permettent une écriture concise et élégante. Le codage des autres opérations s'en trouve souvent allégé. Cependant cette méthode est moins robuste, une variation dans le codage du domaine peut en effet introduire des modifications dans n'importe quelle partie du code. Elle doit donc être réservée à des programmes de petite taille ou bien à des domaines dont la représentation est figée.

4.7.4 Analogie avec les mathématiques

Transparent

On peut essayer de mettre en parallèle les activités de programmation et les mathématiques.
- Spécifier un programme correspond à poser un problème mathématique.

- Analyser un programme, c'est rechercher une démonstration; en particulier l'analyse récursive correspond à une preuve par récurrence.
- Construire un programme (un algorithme) correspond à établir un théorème par une démonstration particulière. Tout comme un théorème admet en général plusieurs démonstrations, un même problème informatique peut être traité par différents algorithmes.
- Le processus d'exécution du programme sur des données correspond à l'application du théorème à un cas particulier.
- La spécification d'un type abstrait s'apparente à la définition d'une structure (comme par exemple les groupes).
- L'ordinateur effectue les tâches élémentaires de calcul comme l'étudiant qui applique la méthode vue en cours pour résoudre son exercice; le programmeur s'apparente lui au mathématicien qui pose les définitions et établit les théorèmes.
- Les différents langages de programmation peuvent être mis en correspondance avec les différentes écoles mathématiques (Bourbaki).
- Les bibliothèques de programmes peuvent être vues comme des théorèmes admis que l'on utilise sans en connaître la démonstration.

Fin 4.54

Comment un mathématicien choisit-il de poser une nouvelle définition ? C'est qu'il a identifié des propriétés importantes partagées par un grand nombre d'objets, et “qui vont bien ensemble” : on peut démontrer des théorèmes beaux et profonds sur les objets possédant ces propriétés. Si les définitions sont absentes ou mal choisies, les démonstrations deviennent beaucoup plus ardues et illisibles. De même, un type abstrait bien choisi peut vous faciliter grandement la vie en vous permettant de faire un seul raisonnement restant valable pour un grand nombre d'objets, et en conduisant à des programmes naturels, élégants et lisibles ; tandis qu'un type abstrait mal choisi peut vous compliquer inutilement la vie.

La conception d'algorithme pour résoudre un problème est similaire à la démonstration de théorèmes en mathématiques, et vous avez déjà commencé à apprendre cette démarche. En revanche on ne vous a jamais demandé en maths d'inventer vous-mêmes vos propres définitions : la conception de types bien choisis vous demande une démarche nouvelle.

Transparent

Cours n^o 5
Tri et complexité

Le tri est une opération importante en informatique, en effet elle permet d'améliorer l'efficacité de nombreuses opérations ayant trait à la manipulation d'un ensemble de données par exemple : retrouver une donnée dans un ensemble, tester l'égalité de deux ensembles .... Le problème du tri va également nous permettre d'aborder la question essentielle de la complexité des algorithmes.

Fin 5.0

5.1 Un algorithme de tri

Transparent

Problème : Construire une fonction qui à une séquence d'entiers associe la séquence formée des mêmes éléments mais triés par ordre croissant.

SPÉCIFICATION:
`tri_seq` (`s:sequence_entiers`) $\rightarrow$ `sequence_entiers`   { `tri_seq(s)`est la séquence triée par ordre croissant formée des éléments de s}

Fin 5.1

Transparent

Analyse :
Le type séquence étant défini récursivement, on procède à une analyse récursive. On regarde de quelle façon la séquence s a été construite.

- Si s correspond à la séquence vide, le résultat est la séquence vide.
- si s n'est pas vide alors elle a été construite par adjonction d'un élément à gauche d'une séquence. On essaye de satisfaire une équation récursive de la forme :

  $$\texttt{tri\_seq}(s) = g(s, \texttt{tri\_seq}(\texttt{reste}(s)))$$

  Si on a déjà trié le reste de la séquence s, il ne reste plus qu'à insérer le premier élément de s au bon endroit. L'équation récursive peut donc s'écrire sous la forme :

  $$\texttt{tri\_seq}(s) = \texttt{insertion}(\texttt{premier}(s), \texttt{tri\_seq}(\texttt{reste}(s)))$$

Fin 5.2

Transparent

RÉALISATION:
tri_seq : (s) $\mapsto$ **si** s = seq_vide   **alors** : s   **sinon** : `insertion(premier(`s`), tri_seq(reste(`s`)))`

Il reste cependant à résoudre le problème de l'insertion d'un élément dans une liste triée.

SPÉCIFICATION:
`insertion (e:entier; s: sequence_entiers)` $\rightarrow$ `sequence_entiers`
{ `insertion(e,s)`insère e dans la séquence s de façon à ce que le résultat reste trié}

Fin 5.3

Analyse :
Il va falloir comparer l'élément à insérer avec ceux de la séquence où l'on doit l'insérer. On regarde comment la séquence a été construite.

- Si s correspond à la séquence vide, le résultat est la séquence réduite à l'élément à insérer.
- si s n'est pas vide alors elle a été construite par adjonction d'un élément à gauche d'une séquence. On essaye de satisfaire une équation récursive de la forme :

$$\texttt{insertion}(e, s) = g(e, s, \texttt{insertion}(e, \texttt{reste}(s)))$$

Transparent

Comme on sait que la séquence s est triée, si $e \leq \texttt{premier}(s)$, il suffit de rajouter e au début de s pour conserver une séquence triée. Dans ce cas on a :

$$\texttt{insertion}(e, s) = \texttt{ajout\_début}(e, s)$$

Sinon, il faut insérer e à un autre endroit de la séquence. Dans ce cas, le premier élément du résultat correspond au premier élément de s. On obtient donc :

$$\texttt{insertion}(e, s) = \texttt{ajout\_début}(\texttt{premier}(s), \texttt{insertion}(e, \texttt{reste}(s)))$$

Fin 5.4

Transparent

RÉALISATION:

```
insertion : (e,s) ↦
  si s = seq_vide
     alors : ajout_debut(e, s)
     sinon :  soit  p = premier(s),
                    si e ≤ p
                       alors : ajout_debut(e,s)
                       sinon : ajout_debut(p, insertion(e,reste(s)))
```

Fin 5.5

Transparent

Codage en CAML :

```
let rec insertion e s =
        match s with
            [] -> [e]
          | pr::reste -> if e <= pr
                         then e::s
                         else pr::(insertion e reste);;
```

```
let rec tri_seq s =
        match s with
           [] -> []
         | pr::reste -> (insertion pr (tri_seq reste));;
```

Fin 5.6

Remarque : Cette méthode de tri qui découle tout naturellement de la récursivité structurelle de la séquence, est connue sous le nom de *tri par insertion*. Elle n'est cependant pas la seule façon de procéder pour trier une séquence. Dans la suite nous étudions une autre méthode.

Transparent

5.2 Notion de complexité

L'exécution d'un algorithme entraine la création d'un processus qui consomme des ressources. Il peut être intéressant d'essayer d'évaluer ces ressources consommées lors de l'exécution. On parle d'*analyse de la complexité* d'un algorithme.

Plusieurs critères sont généralement pris en compte. On cherche à évaluer :

– le coût en temps (*i.e.* le nombre d'opérations nécessaires)
– le coût en espace (*i.e.* la quantité de mémoire nécessaire)

En fait ces deux critères ne sont pas totalement indépendants.

Fin 5.7

5.2.1 Principes de base sur la complexité

Les quantités de ressources consommées par un algorithme sont généralement *fonction de la taille des données* sur lesquelles l'algorithme est appliqué. Une analyse de complexité consiste à essayer de caractériser de telles quantités *en fonction de la taille des données.*

Par ailleurs :

– Evaluer le côut exact est difficile.
– Certaines opérations sont réalisées plus souvent que d'autres, dont le coût devient négligeable par rapport au coût total.

Transparent

C'est pourquoi, lors d'une analyse de complexité, on s'intéresse aux opérations les plus significatives, *i.e.* celles dont le nombre évolue de façon significative lorsque la taille des données varie.

Pour pouvoir comparer des algorithmes, on cherche généralement à exprimer un *ordre de grandeur* de la croissance des ressources consommées par rapport à la taille des données.

Fin 5.8

5.2.2 Ordres de grandeur

Si f est une fonction caractérisant le coût d'un algorithme en fonction de la taille n des données, on s'intéresse à la façon dont croît $f(n)$ lorsque n croît. Plus précisément, on cherche généralement à montrer que $f(n)$ ne croit pas plus vite qu'une autre fonction $g(n)$.

Transparent

Du point de vue mathématique, on dit que la fonction f est *dominée asymptotiquement* par la fonction g ce qui se note $f = O(g)$ et qui signifie :

$$f = O(g) \quad ssi \quad \exists k, \exists n_0, \forall n, \quad n > n_0 \Rightarrow f(n) \leq k.g(n)$$

Exemples :

Croissance	**Ordre de grandeur**
linéaire	$O(n)$
quadratique	$O(n^2)$
polynomiale	$O(n^p)$
exponentielle	$O(p^n)$
logarithmique	$O(log(n))$

Fin 5.9

Transparent

Pour se faire une idée (d'après *Types de données et Algorithmes* Ch. Froidevaux, M.C. Gaudel et M. Soria. Mc Graw Hill (1990))

taille	Complexité						
	1	$log_2(n)$	n	$n.log_2(n)$	n^2	n^3	2^n
$n = 10^2$	$1\mu s$	$6,6\mu s$	$0,1ms$	$0,6ms$	$10ms$	$1s$	$4.10^{16}a$
$n = 10^3$	$1\mu s$	$9,9\mu s$	$1ms$	$9,9ms$	$1s$	$16,6mn$	∞
$n = 10^4$	$1\mu s$	$13,3\mu s$	$10ms$	$0,1s$	$100s$	$11,5j$	∞
$n = 10^5$	$1\mu s$	$16,6\mu s$	$0,1s$	$1,6s$	$2,7h$	$31,7a$	∞
$n = 10^6$	$1\mu s$	$19,9\mu s$	$1s$	$19,9s$	$11,5j$	$31700a$	∞

Fin 5.10

5.2.3 Étude du coût de l'algorithme de tri par insertion

Pour trier une séquence s de n éléments, l'algorithme revient à :
trier une séquence de longeur $n - 1$ (le reste de s)
+ insérer le premier élément dans cette séquence triée de $n - 1$ éléments
Ceci revient donc à :
insérer un élément dans une séquence triée de $n - 1$ éléments
+ insérer un élément dans une séquence triée de $n - 2$ éléments
+ ...
+ insérer un élément dans une séquence triée de 2 éléments
+ insérer un élément dans une séquence triée de 1 élément

Transparent

Cout d'une insertion dans une séquence :

L'opération d'insertion d'un élément dans une séquence oblige à comparer l'élément à insérer avec tous ceux de la séquence. Pour une séquence de p éléments :

Meilleur cas : l'élément est plus petit que le premier élément de la séquence : une seule comparaison.

Cas le pire : l'élément est plus grand que tous les éléments de la séquence : p comparaisons

En moyenne : on peut admettre que le nombre moyen de comparaisons effectuées est de l'ordre de $p/2$.

Fin 5.11

Transparent

Coût total d'un tri par insertion :

Pour une séquence de n éléments.

Meilleur cas : $1 + 1 + 1 + 1 + \ldots + 1$ ($n - 1$ fois)
donc un coût en $O(n)$
Cas le pire : $(n-1) + (n-2) + \ldots + 1$, soit : $n(n-1)/2$
donc un coût en $O(n^2)$
En moyenne : $(n-1)/2 + (n-2)/2 + \ldots + 1/2$, soit : $n(n-1)/4$
donc un coût en $O(n^2)$

Fin 5.12

5.3 Un autre algorithme de tri : le tri rapide (ou tri pivot)

5.3.1 Principe général :

1. Choisir un élément de la séquence (le premier par exemple) nommé *le pivot*
2. Partager la séquence initiale s en deux séquences s_P et s_G contenant les autres éléments de s qui sont plus petits (resp. plus grands) que la valeur du pivot.
3. Trier successivement les deux séquences obtenues :
   $s_P \quad \to t_P$
   $s_G \quad \to t_G$
4. renvoyer la séquence réunissant dans l'ordre : $t_P +$ *le pivot* $+ t_G$

Transparent

Spécification:
`tri_rapide (s: sequence_entiers)` $\to$ `sequence_entiers`
{ `tri_rapide(s)`est la séquence triée formée des éléments de s}

Fin 5.13

5.3.2 Le tri rapide : partage par rapport à une valeur pivot

Transparent

Spécification:
`partage` (p : `entier`; s : `sequence_entiers`) $\to$ `sequence_entiers` $\times$ `sequence_entiers`
{ renvoie (s_1, s_2) tel que s_1 (resp. s_2) contient tous les éléments de s qui sont plus petits (resp. plus grands) que p dans la séquence s}

Fin 5.14

Analyse :
On raisonne sur la structure de la séquence s.
- Si s correspond à la séquence vide, le résultat est une paire de séquences vides.
- si s n'est pas vide alors elle a été construite par adjonction d'un élément à gauche d'une séquence. On essaie de satisfaire une équation récursive de la forme :

  $$\texttt{partage}(p, s) = g(p, s, \texttt{partage}(p, \texttt{reste}(s)))$$

  Si on a déjà partagé le reste de la séquence s, il ne reste plus qu'à examiner le premier élément de s. Selon qu'il est inférieur ou supérieur à la valeur du pivot on l'ajoutera en tête de la première ou de la seconde liste caractérisant $partage(p, reste(s))$

Transparent

RÉALISATION:

partage : (pivot,s) $\mapsto$
  si s = seq_vide
    alors : (s,s)
    sinon : **soit** p = premier(s),
      (s1,s2) = partage(pivot,reste(s)),
      si p $\leq$ pivot
        alors : (ajout_debut(p,s1),s2)
        sinon : (s1,ajout_debut(p,s2))

Fin 5.15

Transparent

RÉALISATION:

tri_rapide : (s) $\mapsto$
  si s = seq_vide
    alors : s
    sinon : **soit** pivot = premier(s),
      (s1,s2) = partage(pivot,reste(s)),
      concat(tri_rapide(s1),ajout_debut(pivot,tri_rapide(s2)))

Fin 5.16

Transparent

Codage en CAML :

```
let rec partage pivot s = match s with
          [] -> ([],[])
        | p::reste -> let (s1,s2) = (partage pivot reste) in
                          if p <= pivot then (p::s1, s2)
                          else (s1, p::s2);;

let rec tri_rapide s = match s with
          [] -> []
        | pivot::reste -> let (s1,s2) = (partage pivot reste) in
                          (tri_rapide s1) @ (pivot::(tri_rapide s2));;
```

Fin 5.17

5.3.3 Complexité de l'algorithme du tri rapide

On s'intéresse toujours au nombre de comparaisons. Celles-ci ont lieu lors du partage de la séquence par rapport au pivot.

- le coût du partage d'une liste de n éléments est n comparaisons.
- le coût du tri d'une liste de n éléments est donc de
    le coût du partage de s en (s_1, s_2) (*i.e.* $n-1$ comparaisons)
  + le coût du tri de s_1
  + le coût du tri de s_2
  Ce coût dépend des longueurs de s_1 et s_2.

Cas le pire

Transparent

A chaque partage, l'une des deux séquences est vide :

```
    tri_r([1,2,3,4,5])
       /        \
tri_r([])    tri_r([2,3,4,5])
                /        \
         tri_r([])    tri_r([3,4,5])
                         /        \
                  tri_r([])    tri_r([4,5])
                                  /        \
                           tri_r([])    tri_r([5])
                                           /        \
                                    tri_r([])    tri_r([])
```

Fin 5.18

Le nombre de comparaisons est de :

$$(n-1)+(n-2)+\ldots+1=n(n-1)/2$$

La complexité au pire est donc en $O(n^2)$

Cas le meilleur

Transparent

Les séquences s_1 et s_2 sont toujours d'égale longueur :

```
                          tri_r([4,2,6,3,5,1,7])
                 /                                    \
         tri_r([2,3,1])                          tri_r([6,5,7])
         /           \                           /           \
   tri_r([1])     tri_r([3])               tri_r([5])     tri_r([7])
    /      \       /      \                 /      \       /      \
tri_r([]) tri_r([]) tri_r([]) tri_r([]) tri_r([]) tri_r([]) tri_r([]) tri_r([])
```

Fin 5.19

Le nombre de comparaisons est de :

$$(n-1)+2*[(n-1)/2-1]+4*[(n-1)/4-1]+\ldots$$

La profondeur de l'arbre est de l'ordre de $log_2(n)$.
La complexité dans le meilleur des cas est donc en $O(nlog_2(n))$

Remarque : La complexité en moyenne est aussi en $O(nlog_2(n))$.

5.4 Complexité en espace

Toute donnée nécessite de l'espace mémoire pour sa représentation.

- Il faut gérer les liens entre identificateurs et valeurs (les environnements)
- Chaque définition locale nécessite la création d'un nouvel environnement (donc des ressources mémoires)
  - définition explicites (**soit** $x = <exp>$, ...)
  - définition implicite (fonction)

Si l'écriture de fonctions récursives permet de donner des expressions simples et concises, l'évaluation d'une telle fonction peut parfois conduire à un processus compliqué.

Transparent

Exemple : la fonction de Fibonacci

RÉALISATION:	
`fib`(n) :	**selon le cas** n
	$n = 0 : 0$
	$n = 1 : 1$
	sinon: $\texttt{fib}(n-1) + \texttt{fib}(n-2)$

Fin 5.20

Lorsque l'on veut visualiser l'evolution de la pile des évaluations en attente on obtient :

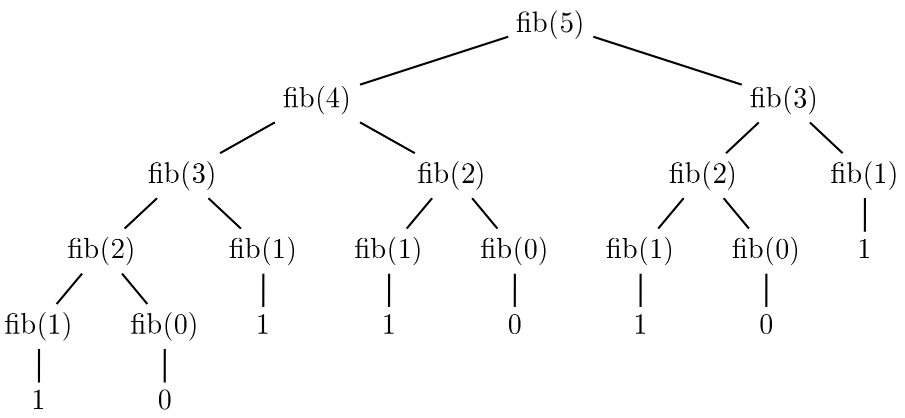
Transparent

```
fib(5)
 fib(4) + fib(3)
  (fib(3) + fib(2)) + fib(3)
   ((fib(2) + fib(1)) + fib(2)) + fib(3)
    (((fib(1) + fib(0)) + fib(1)) + fib(2)) + fib(3)
     (((1 + fib(0)) + fib(1)) + fib(2)) + fib(3)
      ...
 3 + fib(3)
  3  + (fib(2) + fib(1))
   3  + ((fib(1) + fib(0)) + fib(1))
    3  + ((1 + 0) + 1)
      ....
5
```

Fin 5.21

Chaque appel mis en attente nécessite de l'espace mémoire. Ici, le nombre d'additions total croit de façon exponentielle. On peut s'en convaincre à l'aide de l'arbre des appels successifs qui met bien en évidence le fait que la définition de la fonction de fibonacci est doublement récursive.

Transparent



Fin 5.22

Le compromis temps/espace
Une autre façon de calculer la fonction de Fibonacci est d'utiliser une fonction auxilliaire introduisant deux paramètres supplémentaires jouant le rôle d'accumulateurs, mémorisant les deux dernières valeurs calculées de la fonction de Fibonacci.

L'idée est d'avoir une fonction `fib_aux` de signature $(n : \texttt{entier}^*_+ ; a, b : \texttt{entier}_+) \to \texttt{entier}_+$ telle que `fib_aux`(n, a, b) calcule le n-ème terme d'une suite de Fibonacci généralisée définie par

$$u_0 = a,\ u_1 = b\ , u_n = u_{n-1} + u_{n-2}$$

La suite de Fibonacci usuelle est un cas particulier de celle-ci lorsque $a = 0$ et $b = 1$ et on remarque que le n-ème terme d'une telle suite de Fibonacci est égale au $n - 1$-ème terme d'une suite de Fibonacci dont la valeur en 0 est b et la valeur en 1 est $a + b$.

On obtient la réalisation suivante :

Transparent

Réalisation:		
`fib` : (n) ↦	**si** n=0 **alors**: 0	
	sinon: `fib_aux`(n,0,1)	
`fib_aux` : (n,a,b) ↦		**si** n=1 **alors**: b
		sinon: `fib_aux`((n-1),b,(a+b))

Fin 5.23

La fonction `fib_aux` a l'avantage d'être *récursive terminale* c'est-à-dire qu'au moment où se fait l'appel récursif, aucun calcul n'est en attente et le résultat attendu est directement le résultat de l'appel récursif. Ceci permet une compilation plus efficace en espace.

Transparent

Si l'on trace l'évolution de la pile des buts on obtient à présent :

```
fib(5)
 fib_aux(5,0,1)
 fib_aux(4,1,1)
 fib_aux(3,1,2)
 fib_aux(2,2,3)
 fib_aux(1,3,5)
5
```

Fin 5.24

On constate donc que l'introduction de paramètres supplémentaires peut permettre de gagner du temps. Ici, paradoxalement, on constate que cela fait aussi gagner de l'espace car :

- La complexité en temps est maintenant en $O(n)$
- La complexité en espace est aussi en $O(n)$ (voir constante, en cas d'optimisation de l'appel récursif terminal).

Transparent

Cours n^o 6
Séquences génériques

Il est parfois utile de manipuler des données formées de plusieurs objets de même nature comme un jeu de carte, un paquet de livres. Si le nombre maximal d'objets est connu à l'avance, alors on pourra utiliser les domaines produits pour représenter ces objets. Si le nombre est grand ou variable, la représentation par des domaines produits n'est pas commode, elle devient impossible si le nombre d'éléments à traiter n'est pas connu à l'avance. Dans le chapitre 3 nous avions introduit le domaine des séquences d'entiers. Dans ce chapitre, nous montrons que ce domaine peut se voir comme une genéralisation des domaines structurés introduits au chapitre 4. Nous montrons comment manipuler des séquences d'objets de n'importe quelle nature. Enfin, nous introduisons la notion de fonction qui prend en paramètre une autre fonction et illustrons cette notion par la construction d'opérations génériques sur les séquences.

Fin 6.0

6.1 Définition récursive de domaines

Le cours 3 a montré qu'une définition de fonction pouvait être récursive. Dans ce cas la valeur de la fonction appliquée à un argument peut s'obtenir à l'aide de la valeur de cette même fonction appliquée à une autre valeur d'argument. Dans ce chapitre nous allons voir que ce principe peut également être mis en œuvre pour définir de nouveaux domaines.

Définir récursivement un domaine permet notamment de caractériser un ensemble de valeurs par une définition en *compréhension* (par opposition à une définition en *extension*). Ceci s'avère indispensable dès lors que le domaine comporte un nombre infini de valeurs.

Une définition récursive de domaine a de plus l'intérêt d'être **constructive**, *i.e.* elle décrit non seulement les différentes valeurs du domaine mais aussi comment celles-ci peuvent s'obtenir. En fait une telle définition énumère les différentes façons de *produire* les valeurs du domaine par le biais :

- des constantes du domaine
- des opérations de construction

La différence par rapport à une définition non récursive est que certaines opérations de construction utilisent des valeurs du domaine que l'on cherche à définir (des valeurs que l'on sait déjà produire) pour produire de nouvelles valeurs de ce domaine.

6.1.1 Quelques exemples de définitions récursives de domaines

Transparent

Exemple 1: la spécification suivante constitue une définition constructive du domaine des entiers naturels positifs $\mathbb{N}$:

SPÉCIFICATION:
{Domaine}
`entier`$_+$
{Constantes}
`0` : `entier`$_+$
{Fonction de construction}
`plus1` (n: `entier`$_+$) $\rightarrow$`entier`$^*_+$
{Fonctions d'accès}
`moins1` (n : `entier`$^*_+$) $\rightarrow$`entier`$_+$

Fin 6.1

Cette définition est *bien fondée* dans la mesure où pour tout entier $n \in \mathbb{N}$ on a bien soit $n = 0$, soit $n = (n-1)+1$. Elle fait clairement apparaître le fait que tout entier n peut s'obtenir à partir de la constante 0 par la succession de n applications successives de l'opération `plus1`.

Exemple 2 : Dans le chapitre 3 nous avons introduit le domaine des séquences d'entiers. Nous pouvons maintenant reconnaître dans `ajout_debut` une opération de construction récursive d'une séquence et dans `premier` et `reste` les deux opérations d'accès correspondantes.

Transparent

SPÉCIFICATION:
{Domaine}
`sequence_entiers`
{Constantes}
`seq_vide`:`sequence_entiers`
{Fonction de construction}
`ajout_debut` (n: `entier`;s:`sequence_entiers`) $\rightarrow$ `sequence_entiers`*
{Fonctions d'accès}
`premier` (s : `sequence_entiers`*) $\rightarrow$ `entier`
`reste` : (s:`sequence_entiers`*) $\rightarrow$`sequence_entiers`

Fin 6.2

Exemple 3 : Le même principe peut être utilisé pour donner une définition constructive du domaine des chaînes de caractères. Dans ce cas, cette définition exprime que toute chaîne peut s'obtenir à partir d'une chaîne *vide* en lui rajoutant successivement tous les caractères qui la composent.

Transparent

SPÉCIFICATION:
{Domaine}
`chaine`
{Constantes}
`chaine_vide` : `chaine`
{Fonction de construction}
`ajout_debut_ch` (c : `caractere`; s: `chaine`) $\rightarrow$ `chaine`*
{Fonctions d'accès}
`premier_ch` (s: `chaine`*) $\rightarrow$ `caractere`
`reste_ch` (s:`chaine`*) $\rightarrow$ `chaine`

Fin 6.3

L'opération ajout_debut_ch a pour effet de construire la chaîne obtenue en rajoutant un nouveau caractère au début d'une chaîne déja existante. Le domaine chaine* représente ici l'ensemble des chaines qui sont différentes de chaine_vide.

Codage CAML

Il est possible de coder en CAML un domaine récursif avec un type de données récursif : les arguments associés à chaque étiquette peuvent mentionner le type que l'on est en train de définir. Ainsi les domaines $\texttt{entier}_+$ et chaine préalablement introduits peuvent se coder de la manière suivante.

Transparent

```
#type entier_plus = Zero | Plus1 of entier_plus;;

#let plus1 n = Plus1(n);;
val plus1 : entier_plus -> entier_plus = <fun>

#let moins1 n = match n with
     Plus1(m) -> m
   | _        -> failwith "entier nul";;
val moins1 : entier_plus -> entier_plus = <fun>
```

Fin 6.4

Transparent

```
#type chaine = Vide | AjoutDebut of char * chaine;;

#let chaine_vide = Vide;;
val chaine_vide : chaine = Vide

#let ajout_debut_ch c s = AjoutDebut(c,s);;
val ajout_debut_ch : char -> chaine -> chaine = <fun>

#let premier_ch s = match s with
    AjoutDebut(c,_) -> c
  | _ -> failwith "chaine vide";;
val premier_ch : chaine -> char = <fun>

#let reste_ch s = match s with
    AjoutDebut(_,s') -> s'
  | _ -> failwith "chaine vide";;
val reste_ch : chaine -> chaine = <fun>
```

Fin 6.5

6.1.2 Domaines récursifs et fonctions récursives

Lors du chapitre sur les fonctions récursives, nous avons vu que l'analyse récurrente d'une fonction f revenait à identifier d'une part des cas de base, pour lesquels il est possible de donner une expression directe de la valeur de f, et d'autre part des cas généraux, pour lesquels la valeur de la fonction peut se caractériser par le biais d'une équation récursive de la forme :

$$f(x) = g(x, f(h(x)))$$

Lorsque la fonction f a de paramètres autres que le paramètre sur lequel se fait l'analyse récurrente, ceux-ci apparaissent également comme paramètres de g. Si f est une fonction à 2 paramètres x et y et que l'analyse récurrente se fait sur x alors l'équation aura en général la forme :

$$f(x,y) = g(x, y, f(h(x), y))$$

La fonction h traduit alors la façon dont on passe de la donnée x à une donnée "plus petite" $h(x)$ sur laquelle on procède à l'appel récursif $f(h(x))$ et g traduit la façon dont on utilise le résultat de l'appel récursif pour décrire la valeur de $f(x)$.

Lorsque la fonction f travaille sur des donnés d'un domaine dont la définition est récursive on est généralement conduit à une telle analyse récurrente. Dans ce cas, la relation d'ordre sur les valeurs du domaine correspond tout simplement à l'ordre de construction des valeurs du domaine, *i.e.* une valeur est plus petite qu'une autre si elle a été utilisée pour produire la seconde (directement ou indirectement). La réalisation de la fonction se traduit alors généralement par une décomposition par cas dans laquelle :
- les cas de base correspondent aux cas de base de la définition du domaine
- les cas recursifs correspondent aux cas où la valeur considérée a été construite récursivement à partir d'une autre valeur du domaine. Dans ce cas la fonction h correspond généralement à l'opération d'accès permettant de retrouver cette valeur.

Exemples

Transparent

Exemple 1 : Comme déjà vu précédemment, la réalisation de la fonction `puiss`(a, n) revient à distinguer les cas :
- si $n = 0$ alors `puiss`$(a, 0) = 1$
- si $n > 0$ alors `puiss`$(a, n) = n *$ `puiss`$(a,$ `moins1`$(n))$

Fin 6.6

L'équation s'écrit `puiss`$(a, n) = g(a, n,$ `puiss`$(a, h(n)))$

Dans cet exemple la fonction h correspond à la fonction `moins1` alors que la fonction g correspond au produit de a par son dernier argument.

Exemple 2 : **Problème :** Définir la fonction `long_ch` qui calcule la longueur d'une chaîne de caractères.

Transparent

SPÉCIFICATION:
`longeur_ch` (ch : `chaine`) $\rightarrow$ `entier`
{ calcule la longeur de la chaine ch}

Analyse :
On sait qu'une chaîne peut être soit vide soit construite par ajout d'un caractère au début d'une chaîne. Dans le premier cas la longueur de la chaîne est 0. Dans le second cas il y a exactement un caractère de plus que dans la chaîne à partir de laquelle elle a été construite.

Fin 6.7

On en déduit la réalisation :

Transparent

RÉALISATION:	
`longueur_ch` : (ch) $\mapsto$	**si** ch = `chaine_vide`
	alors : 0
	sinon : 1+ `longueur_ch`(`reste_ch`(ch))

Fin 6.8

Ici c'est la fonction d'accès `reste_ch` qui joue le rôle de la fonction h. La chaîne sur laquelle s'effectue l'appel récursif a bien été construite avant celle sur laquelle s'effectue l'appel initial et est donc bien inférieure au sens de l'ordre de construction des valeurs du domaine.

Codage CAML

Le codage CAML suit la réalisation, les fonctions d'accès peuvent être remplacées par l'utilisation du filtrage.

Transparent

```
let rec longueur_ch s = match s with
    Vide              -> 0
  | AjoutDebut(_,r)   -> 1 + longueur_ch r;;
```

Fin 6.9

6.2 Définition d'un domaine générique de séquence

Le principe de construction du domaine des chaînes de caractères est caractéristique des structures séquentielles et peut être repris pour définir d'autres domaines similaires.

Une *séquence* est un ensemble d'éléments de même type et placés dans un ordre précis. On retrouve de telles structures sous d'autres appellations comme les *listes*, les *suites*. Elles sont très utilisées en informatique :
- chaînes de caractères
- fichiers
- suites de valeurs
- ensembles

La structure de séquence sous-entend un mode séquentiel pour accéder aux éléments qui la composent et c'est pourquoi nous préférons conserver ici le terme *séquence* plutôt que celui de liste.

Il existe aussi des variantes possédant des propriétés supplémentaires, comme les *files* ou les *piles*.

6.2.1 Séquences génériques dans le langage de description

Afin de ne pas redéfinir un nouveau domaine pour chaque catégorie de séquence, on peut faire **abstraction** du domaine des éléments et proposer une définition plus "générale", un peu comme nous l'avions fait pour représenter un produit générique de domaine.

Transparent

SPÉCIFICATION:
{Domaine}
`sequence` {paramétré par `<type_Element>`}
{Constantes}
`seq_vide` : `sequence`
{Fonction de construction}
`ajout_debut` (e:<type_Element>; s:`sequence`) → `sequence`*
{Fonctions d'accès}
`premier`(s:`sequence`*) → `<type_Element>`
`reste`(s:`sequence`*) → `sequence`

Fin 6.10

Dans cette définition `<type_Element>` représente un *paramètre de domaine*, susceptible de prendre n'importe quelle valeur de domaine. Le domaine ainsi spécifié est dit *générique* ou *polymorphe.*

Dans notre langage de description, on s'autorise la spécification de domaines par *spécialisation de domaines génériques.* Pour cela, il suffit de donner une valeur précise aux paramètres de domaines. Ainsi, pour définir un domaine correspondant à des séquences d'éléments d'un domaine particulier il suffit d'utiliser une définition de la forme :

RÉALISATION:
{Domaine} <nomType> = `sequence` de <type_Elément>

Transparent

Exemple :

RÉALISATION:
{Domaine} `sequence_entiers` = `sequence` de `entier`
{Domaine} `chaine` = `sequence` de `caractere`

L'intérêt d'utiliser des définitions génériques de domaines est manifeste dès lors que l'on cherche à définir des fonctions dont le résultat est indépendant de valeurs de tels paramètres de types.

Fin 6.11

Exemple le principe de calcul de la longueur d'une chaîne de caractères peut être repris pour définir une fonction générale calculant la longueur d'une séquence.

Transparent

SPÉCIFICATION:
`longueur` (s : `sequence`) → `entier`
{ calcule le nombre d'éléments de la séquence s }

Remarque : Dans l'énoncé de la spécification, lorsque le paramètre de type n'intervient pas en tant que tel, nous allègerons la notation et noterons simplement `sequence` au lieu de `sequence de <e>`.

Fin 6.12

Transparent

Le principe de la réalisation est rigoureusement identique à celui du calcul de la longueur d'une chaine, soit :

RÉALISATION:
`longueur` : (s) $\mapsto$ **si** s = `seq_vide` **alors** : 0 **sinon** : 1+ `longueur(reste(s))`

Fin 6.13

6.2.2 Mise en œuvre d'un type générique en CAML

Le codage des domaines génériques est facilité par le fait que CAML autorise le polymorphisme. C'est une caractéristique très appréciée du langage CAML qui n'existe malheureusement pas dans tous les langages de programmation. Si ce n'est pas le cas, l'étape de codage peut alors nécessiter de définir un type différent pour chaque instance utilisée du type générique et de recoder différents "clônes" des fonctions génériques pour chacune de ces instances.

En CAML on peut spécifier un paramètre de type à l'aide d'un symbole précédé d'une quote.

Exemple : `'a` représente "un type quelconque a".

La définition d'un type paramêtré s'effectue en énonçant d'abord le paramètre de type, suivi du nom du type. Par exemple pour coder le type générique séquence :

Transparent

Codage en CAML :

```
type 'a sequence =
           Seq_Vide
         | AjoutDebut of 'a * 'a sequence ;;

(* opération de construction *)
let ajout_debut e s = AjoutDebut(e,s);;

(* opérations d'accès *)
let premier l = match l with
       AjoutDebut(e,_) -> e
      | _ -> failwith "sequence vide" ;;
let reste l = match l with
       AjoutDebut(_,s) -> s
      | _ -> failwith "sequence vide" ;;
```

Fin 6.14

Transparent

Le codage de la fonction `longueur` se résume alors simplement à :

Codage en CAML :

```
#let rec longueur s =
      match s with
            Seq_Vide -> 0
          | AjoutDebut(_,reste) -> 1+(longueur reste) ;;

longueur : 'a sequence -> int = <fun>
```

Fin 6.15

6.3 Le type CAML prédéfini `'a list`

En pratique, comme la notion de séquence est très utilisée, on trouve dans beaucoup de langages un moyen de représenter facilement ce type de structure. En CAML il existe sous la forme d'un type générique appellé : `'a list`

Pour ce type la séquence vide est représentée par la constante `[]`.

```
#[];;
- : 'a list = []
```

Les notations et opérateurs de CAML pour les listes qui été introduits au chapitre 3 peuvent également s'utiliser sur des listes d'éléments quelconques. Tous les éléments de la liste doivent avoir le même type.

Transparent

Exemples :

```
#1::[];;
- : int list = [1]

#1:: 2 :: 3 ::[];;
- : int list = [1; 2; 3]

#"juste" :: "quelques" :: "mots" :: [];;
- : string list = ["juste"; "quelques"; "mots"]

# ["juste"; 2; "mots"];;
This expression has type int but is here used with type string
```

Fin 6.16

Le codage de la fonction longueur sur le type `'a list` donne donc dans ce cas :

Transparent

```
#let rec longueur s =
     match s with
        [] -> 0
      | _::reste -> 1+(longueur reste) ;;
val longueur : 'a list -> int = <fun>
```

La fonction primitive `List.length` calcule la longueur d'une liste d'éléments quelconques.

Fin 6.17

Utilisation du type 'a list

Le fait d'utiliser des types génériques peut poser des états d'âme lors de la modélisation d'un problème. Si l'on a besoin de manipuler une structure séquentielle on est amené à choisir entre deux solutions :

– redéfinir un nouveau type à partir de rien (comme nous l'avons fait pour les chaînes de caractères)
– utiliser une instance du type générique séquence (par exemple une séquence de caractères)

A priori, il semble plus simple de s'appuyer sur les types génériques dont l'intérêt est justement d'être réutilisés. Le seul inconvénient dans ce cas est qu'on ne verra pas apparaître au niveau du codage le nom du type initialement retenu (au lieu d'avoir `chaine` on aura `char list`).

Il peut cependant se trouver des situations où l'on est amené à manipuler différents domaines correspondants à des structures de séquences et dont les domaines de base coïncident. Par exemple on peut vouloir représenter simultanément des suites de lettres majuscules, des suites de caractères représentant des chiffres et des suites de caractères spéciaux. Si l'on s'appuie sur le type générique `list`, alors, au niveau du codage, dans tous les cas on est ramené à manipuler des valeurs de type `char list`, ce qui peut donc être l'occasion d'erreurs de type indétectables par le système CAML. On peut dans ce cas décider de coder des types différents pour chaque instance. Tout dépend du niveau d'exigence que le programmeur s'impose au niveau du code produit.

6.4 Exemples de fonctions sur les séquences

Dans cette section nous reprenons les différentes opérations de base sur des structures de séquences qui avaient été introduites au chapitre 3. Nous montrons comment leur spécification peut être généralisée pour s'appliquer à des séquences d'objets quelconques. Nous ne redonnons pas les étapes d'analyse conduisant à la réalisation car elles sont identiques à ce qui avait été fait pour les séquences d'entiers.

6.4.1 Concaténation de séquences

Transparent

Problème : Construire une fonction calculant la séquence obtenue en réunissant deux séquences bout à bout.

SPÉCIFICATION:
`concat` ($s1, s2$: `sequence`) $\rightarrow$ `sequence`
{ calcule la séquence formée des éléments de $s1$ suivis de ceux de $s2$}

Fin 6.18

Transparent

RÉALISATION:
`concat` : (s1,s2) $\mapsto$ **si** $s1$ = seq_vide
alors : $s2$
sinon : `ajout_debut(premier(`$s1$`),concat(reste(`$s1$`),`$s2$`))`

Codage en CAML :

```
#let rec concat s1 s2 =
        match s1 with
              []             -> s2
            | premier::reste -> premier::(concat reste s2);;
val concat : 'a list * 'a list -> 'a list = <fun>
```

Fin 6.19

Remarque : Nous rappelons que cette opération est une opération prédéfinie sur les listes en CAML et correspond à l'opérateur infixe `@`. On peut donc représenter de façon équivalente le résultat de (`concat` s1 s2) par : s1 `@` s2 .

6.4.2 Appartenance d'un élément à une séquence

Transparent

Problème : Construire une fonction qui vérifie si un élément figure bien dans une séquence.

SPÉCIFICATION:
`appartient_seq` (e : `<e>`; s : `sequence de <e>`) $\rightarrow$ `booléen` { vrai ssi $e \in s$}

Fin 6.20

Remarque : Ici, on fait explicitement apparaître `sequence de <e>` car il est important de préciser que le domaine des éléments de s doit être le même que celui de e.

Transparent

RÉALISATION:
`appartient_seq` : $(e, s) \mapsto$ **si** $s =$ `seq_vide` **alors** : `faux` **sinon** : $e =$ `premier`(s) **ou** `appartient_seq`(e,`reste`(s))

Codage en CAML :

```
#let rec appartient_seq e s=
        match s with
              []   -> false
            | p::r -> e = p or (appartient_seq e r);;

val appartient_seq : 'a -> 'a list -> bool = <fun>
```

Cette fonction est prédéfinie dans la bibliothèque `List` de CAML et porte le nom `mem`.

Fin 6.21

6.4.3 Inversion de séquence

On vérifie aisément que l'inversion de séquence telle qu'elle a été définie au chapitre 3 se généralise à des séquences génériques. La spécification de cette fonction devient :

Transparent

`inv_seq` (s : `sequence`) $\rightarrow$ `sequence`
{ calcule la séquence inverse de s}

Le type CAML de cette fonction est :

```
val inv_seq : 'a list -> 'a list = <fun>
```

Cette fonction est prédéfinie dans la bibliothèque `List` de CAML et porte le nom de `rev`.

Fin 6.22

6.4.4 Méthode de tri générique

Les fonctions de tri que nous avons étudiées dans le chapitre 5 se généralisent au tri de séquences d'éléments dans un domaine `<e>` arbitraire. Pour cela, il faut se donner une fonction de comparaison des éléments de `<e>`. On suppose définie une fonction de comparaison `inf`($x1, x2$: `<e>`) $\rightarrow$ `booleen`. Nous reprenons les spécifications et réalisations du tri par insertion mais le tri rapide peut également être étendu au cas générique.

Transparent

Spécification:

`tri` (s : `sequence de <e>`) $\rightarrow$ `sequence de <e>`
{ calcule la séquence obtenue à partir de s en triant les éléments à l'aide de la fonction de comparaison `inf`}
`insertion` (x : `<e>`; s : `sequence de <e>`) $\rightarrow$ `sequence de <e>`
{ calcule la séquence triée obtenue en insèrant x dans la séquence s supposée triée}

Fin 6.23

La réalisation est exactement la même que dans le cas des séquences d'entiers sauf que le test $e < p$ est remplacé par l'utilisation de la fonction de comparaison spécifique `inf`(e,p).

Transparent

Réalisation:

`tri` : (s) $\mapsto$ **si** s = seq_vide **alors**: s
sinon: `insertion`(`premier`(s), `tri`(`reste`(s)))
`insertion` : (e,s) $\mapsto$
si s = `seq_vide`
alors : `ajout_debut`(e,s)
sinon : **soit** p = `premier`(s),
si `inf`(e,p) **alors**: `ajout_debut`(e,s)
sinon: `ajout_debut`(p, `insertion`(e,`reste`(s)))

Fin 6.24

6.5 Fonctions d'ordre supérieur

6.5.1 Données et traitements

Les chapitres précédents ont mis en évidence le rôle central de **l'abstraction** dans la modélisation des problèmes.

Nous avons surtout mis l'accent sur l'abstraction au niveau des données. Les langages fonctionnels permettent généralement de renforcer le rôle de l'abstraction au niveau des traitements.

Les modèles de programmation "classiques" distinguent les données et les opérations qui manipulent ces données. Ces dernières correspondent à du code exécuté par la machine.

Dans les langages fonctionnels, la notion de fonction joue un rôle de premier plan. Les fonctions servent à décrire des opérations sur les données mais peuvent également être vues elle-mêmes comme des données et être manipulées par d'autres opérations.

Les fonctions sont considérées comme des **objets de *première classe***

Puisque les fonctions sont considérées comme des valeurs ordinaires, on peut les utiliser de la même façon que l'on utilise les valeurs d'autres types.

En particulier :
- On peut les utiliser comme paramètres de fonctions
- On peut les utiliser comme résultats de fonctions (mais nous n'aborderons pas cet aspect dans ce cours).

On parle alors de **fonctions d'ordre supérieur**

Dans ce chapitre nous enrichissons notre langage de description pour que les fonctions puissent accepter d'autres fonctions comme argument. Nous montrons quelques exemples de manipulation de fonctions d'ordre supérieur. Nous allons voir comment les fonctions d'ordre supérieur nous permettent de décrire des traitements génériques sur les séquences qui pourront être réutilisés dans de nombreuses situations.

6.5.2 Exemple

Si une séquence est utilisée pour représenter un ensemble fini F de valeurs entières, et soit p une propriété sur les entiers, on peut s'intéresser à savoir si F contient une valeur qui vérifie p.

Transparent

Problème : Soit p un prédicat sur les éléments de `<e>`, construire une fonction qui teste si un élément d'une séquence vérifie la propriété p.

SPÉCIFICATION:
`exists` (s : `sequence de <e>`) $\rightarrow$ `booleen`
{ vrai ssi il existe un élément de s qui vérifie p}

Fin 6.25

Analyse :

Le type séquence étant défini récursivement, il faut procéder à une analyse récursive.
- si s correspond à la séquence vide, le résultat est faux,

– si s n'est pas vide alors elle a été construite par adjonction d'un élément à gauche d'une séquence. On essaye de satisfaire une équation récursive de la forme :

$$\texttt{exists}(s) = g(e, s, \texttt{exists}(\texttt{reste}(s)))$$

Ici la fonction g n'a qu'à tester si $p(\texttt{premier}(s))$ est vrai ou si $\texttt{exists}(\texttt{reste}(s))$ On a donc $g(e, s, y) = p(\texttt{premier}(s))$ ou y.

Transparent

RÉALISATION:
`exists` : (s) $\mapsto$ **si** s = `seq_vide` **alors** : `vrai` **sinon** : p(`premier`(s)) ou `exists`(`reste`(s))

```
let rec exists s =
        match s with
              [] -> true
            | premier::reste -> (p premier) or (exists reste);;
```

Fin 6.26

On pourra de manière analogue, spécifier une fonction `pourtout` qui teste si tous les éléments de la séquence vérifient p.

Dans ces fonctions, le prédicat à tester n'intervient que comme un paramètre à tester sur les valeurs de la séquence, le traitement à effectuer est le même que l'on cherche un entier égal à une valeur e particulière (auquel cas on retrouve la fonction `appartient_seq`), un nombre pair ou encore un livre non rendu à la bibliothèque.

Nous allons voir que l'utilisation de fonctions d'ordre supérieur permet de modéliser de telles opérations de manière générique, en prenant comme argument des fonctions qui représentent les traitements à effectuer dans chaque cas. Ces fonctions seront instanciées dans chaque utilisation particulière.

6.5.3 Les fonctions comme paramètre d'autres fonctions

Pour pouvoir considérer les fonctions comme des paramètres, il nous faut étendre notre langage de description de manière à introduire des domaines dont les éléments sont des fonctions.

Domaines de fonctions Une valeur fonctionnelle correspond à un ensemble d'instructions machines (le code de la fonction).

Les fonctions peuvent être classées dans différents domaines suivant les données qu'elles manipulent. Le domaine d'une fonction correspond à ce que nous avons appelé un *profil*.

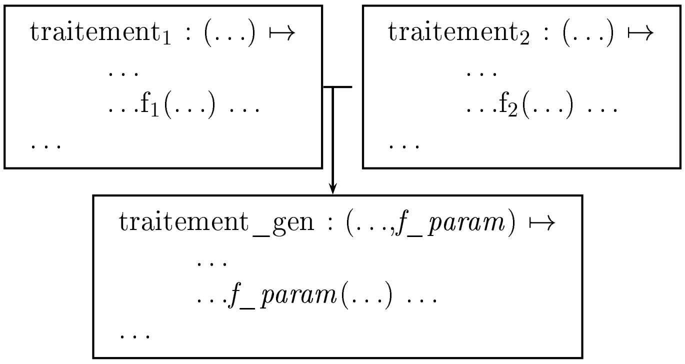
Exemple :

– `entier` $\rightarrow$ `entier`
  représente le domaine des fonctions à valeur entière et à un paramètre entier.
  La fonction qui a un entier x associe l'entier $x+1$ est une valeur particulière de ce domaine.
– (`entier` ; `entier`) $\rightarrow$ `booleen`
  représente le domaine des fonctions qui à deux entiers associent une valeur booléenne.

6.5.4 Schémas généraux

Utiliser des fonctions comme paramètres d'autres fonctions permet de mettre en évidence des **schémas généraux** de traitements qui ne diffèrent que par les fonctions intermédiaires utilisées dans ces traitements.

Transparent



Fin 6.27

Retour sur l'exemple

Nous pouvons réécrire l'exemple de la recherche dans une séquence d'un élément vérifiant une certaine propriété en introduisant le prédicat p comme un paramètre de la fonction :

Transparent

RÉALISATION:
`exists` : (p,s) $\mapsto$ **si** s = `seq_vide` **alors** : `vrai` **sinon** : p(`premier`(s)) `ou` `exists`(p,`reste`(s))

La signature de la fonction est alors :
`exists` : (p:<e> $\rightarrow$ `booleen`; s: `sequence de <e>`) $\rightarrow$ `booleen`

Fin 6.28

Transparent

Le codage CAML suit le même principe :

```
#let rec exists p s =
        match s with
              [] -> true
            | premier::reste -> (p premier) or (exists p reste);;
val exists : ('a -> bool) -> 'a list -> bool = <fun>

#let appartient_seq e s =  let p x = (x=e) in (exists p s);;
val appartient_seq : 'a -> 'a list -> bool = <fun>
```

Fin 6.29

6.5.5 Abstraction d'un schéma type de traitement de séquence

De nombreuses opérations sur les séquences suivent le même schéma d'analyse récursive. On peut introduire des opérateurs qui capturent ces schémas de récurrence.

Transparent

Exemple :

RÉALISATION:

`longueur`	: (s)	$\mapsto$ **si** s=`seq_vide` **alors**:0 **sinon**: 1+`longueur`(`reste`(s))
`somme`	: (s)	$\mapsto$ **si** s=`seq_vide` **alors**:0 **sinon**: `premier`(s)+`somme`(`reste`(s))
`inv_seq`	: (s)	$\mapsto$ **si** s=`seq_vide` **alors**: s **sinon**: `ajout_fin`(`premier`(s),`inv_seq`(`reste`(s)))

Fin 6.30

Un schéma type de traitement de séquence On a toujours le même type de schéma :

$$\texttt{f} : (s) \mapsto \textbf{si } s = \texttt{seq\_vide} \textbf{ alors} : \textit{valeur} \textbf{ sinon} : g(s, f(\texttt{reste}(s)))$$

Il peut être intéressant de définir une fonction `traite_sequence` qui à partir d'une valeur initiale et d'une fonction g engendre la fonction de traitement de séquence correspondante.

Transparent

L'opération `traite_sequence` a pour signature :

$$(v : T; g : (\texttt{sequence}; T) \to T; s : \texttt{sequence}) \to T$$

et est réalisée de la manière suivante :

$$\begin{array}{ll} \texttt{traite\_sequence} : (v, g, s) \mapsto & \textbf{si } s\texttt{=seq\_vide } \textbf{alors}: \text{ v} \\ & \textbf{sinon}: \ g(s, \texttt{traite\_sequence}(v, g, \texttt{reste}(s))) \end{array}$$

Fin 6.31

On pourra alors utiliser cette opération pour réaliser les différentes fonctions sur les séquences :

Transparent

RÉALISATION:

`g_longueur`	:(s,n)	$\mapsto$ 1+n
`longueur`	:(s)	$\mapsto$ `traite_sequence`(0,`g_longueur`,s)
`g_somme`	:(s,n)	$\mapsto$ `premier`(s)+n
`somme`	:(s)	$\mapsto$ `traite_sequence`(0,`g_somme`,s)
`g_inv_seq`	:(s,is)	$\mapsto$ `ajout_fin`(`premier`(s),is)
`inv_seq`	:(s)	$\mapsto$ `traite_sequence`(`seq_vide`,`g_inv_seq`,s)

Fin 6.32

Transparent

Codage en CAML :

```
#let traite_sequence valeur_init g s =
     let rec f s = match s with
                      [] -> valeur_init
                    | (_::s') -> (g s (f s'))
     in f s;;
traite_sequence : 'a -> ('b list -> 'a -> 'a) -> 'b list -> 'a = <fun>

#let g_longueur s x = 1+x;;
#let longueur s = traite_sequence 0 g_longueur s;;
long : 'a list -> int = <fun>
```

Fin 6.33

6.5.6 Transformer les éléments d'une liste

Une autre opération paramétrée utile consiste à appliquer la même opération à tous les éléments d'une séquence.

Cette opération prend en argument une fonction f de T dans T' et une séquence s d'éléments dans le domaine T, elle renvoie une séquence d'éléments de T' obtenus en appliquant f à chaque élément de s.

Transparent

SPÉCIFICATION:
`applique` $(f : T \to T'; s :$ `sequence de` $T) \to$ `sequence de T'`
$\{$ calcule$[f(a_1); \ldots; f(a_n)]$ si $s = [a_1; \ldots; a_n]\}$

RÉALISATION:	
`applique` (f,s) $\mapsto$	**si** s=`seq_vide` **alors**: `seq_vide`
	sinon: `ajout_debut`(f(`premier`(s)),`applique` (f,`reste`(s)))

Fin 6.34

Transparent

Codage en CAML :

```
#let applique f s =
    let rec app_rec s = match s with
            [] -> []
          | a::rest -> f(a) :: (app_rec rest)
    in (app_rec s);;
applique : ('a -> 'b) -> 'a list -> 'b list = <fun>

#let plus1 x = x+1;;
plus1 : int -> int = <fun>
#applique plus1 [1;2;3];;
- : int list = [2; 3; 4]
```

Fin 6.35

La fonction `applique` est prédéfinie en CAML et se nomme `List.map`.

Transparent

Cours n^o 7
Structures d'arbres

La structure d'arbre est très utilisée en informatique, que ce soit pour :
- présenter un ensemble d'objets ou d'informations élémentaires organisé en une structure hiérarchique
- en faciliter l'accès ou la recherche
- modéliser de nombreux problèmes

Fin 7.0

7.1 Introduction - Exemples

Du point de vue informatique les structures d'arbres constituent un complément indispensable de la structure de séquence.

En effet la structure de séquence est une organisation linéaire de l'information où tous les objets sont au même niveau. De nombreuses modélisations se prêtent naturellement à une organisation hiérarchique en plusieurs niveaux.

7.1.1 Exemples de modélisations sous formes d'arbres :

Livre le sommaire d'un livre reflète une hiérarchie :
- chapitres
  - sections
    - sous-sections
      - paragraphes

Fichiers L'organisation des fichiers d'un système d'exploitation en répertoires et sous-répertoires correspond à une structure d'arbre.

Expression arithmétique Une expression arithmétique qui n'est pas une constante se décompose en un opérateur et deux opérandes qui sont eux-mêmes des expressions.

Arbres syntaxiques Une phrase est structurée en plusieurs entités (groupes nominaux, verbe, compléments) qui elles-mêmes peuvent être décomposées.

Arbres de classifications En botanique ou zoologie, on utilise des arbres pour classifier les espèces.

Arbres de jeu La modélisation d'un jeu entre deux adversaires peut se faire à l'aide d'un arbre représentant les coups possibles et leurs ripostes.

Tâches Un raisonnement ou une tâche peut se décomposer en plusieurs sous-tâches sous une forme arborescente.

7.1.2 Représentations de la structure d'arbre

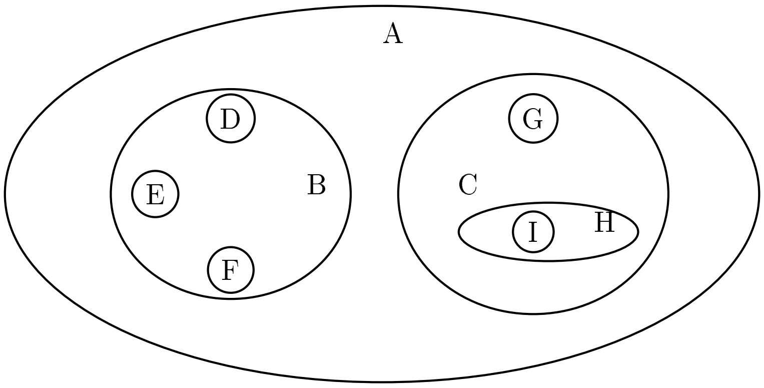
Différentes représentations sont possibles :
- ensembles emboîtés
- graphe
- mise en page indentée
- représentations linéaires avec parenthèses

Chaque type de représentation a ses avantages et ses inconvénients en fonction du point de vue adopté :
- lisibilité pour l'homme (représentations graphiques)/ facilité de mise en œuvre en machine (représentations linéaires, textuelles)
- explicitation des niveaux d'abstraction (ensembles emboîtés)/ des relations entre éléments (graphes)

Quelques exemples de représentation

Transparent

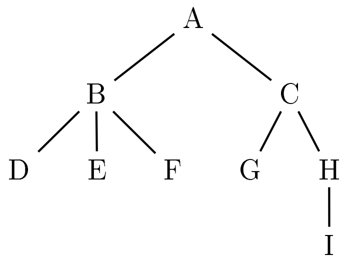
Représentation ensembliste :



Fin 7.1

Transparent

Représentation sous forme d'un graphe:



Fin 7.2

Transparent

Représentation indentée :

```
A
  B
     D
     E
     F
  C
     G
     H
        I
```

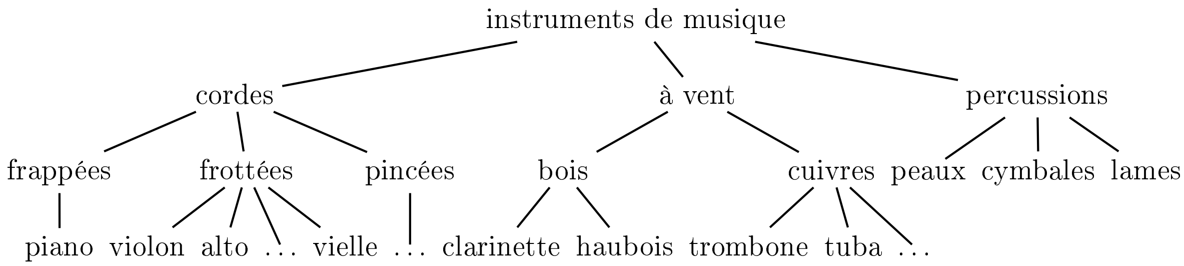
Représentation linéaire préfixée ou postfixée :

A(B(D,E,F),C(G,H(I))) ou ((D,E,F)B,(G,(I)H)C)A

Fin 7.3

Transparent

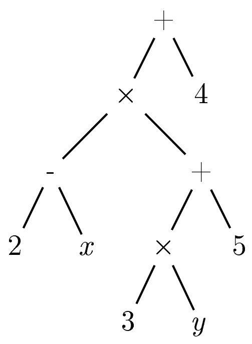
Arbre représentant une classification



Fin 7.4

Transparent

Arbre représentant une expression arithmétique
L'expression $(2 - x) \times (3 \times y + 5) + 4$ peut être représentée par l'arbre :



Fin 7.5

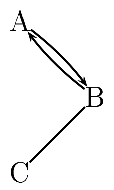
Remarque : les nombres et les variables correspondent aux feuilles de l'arbre alors qu'à l'intérieur on trouve des opérateurs

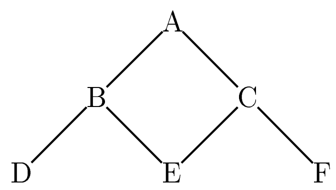
Remarque : Les informaticiens ont pour habitude de représenter les arbres à l'envers

- la racine est en haut
- les feuilles en bas
- les branches sont orientées vers le bas

Transparent

Contre-exemples La structure d'arbre correspond à une hiérarchie stricte. Les structures suivantes ne sont pas des arbres :





Fin 7.6

Ce sont en fait des structures de graphes. Une propriété caractéristique d'un arbre est que tout nœud, à l'exception de la racine, n'a qu'un seul nœud parent.

Remarque : La seconde structure pourrait se représenter sous la forme d'un arbre en dupliquant le nœud E .

7.2 Définitions

7.2.1 Définitions graphe et arbre

Un arbre au sens informatique est un cas particulier de *graphe*.

Graphe Un *graphe* G est composé d'un ensemble (en général fini) S d'objets appelés *sommets* et d'une relation binaire sur S c'est-à-dire d'un ensemble de couples (s_1, s_2) avec $s_1, s_2 \in S$ appelés des *arêtes* ou des *arcs*.

Chemin-longueur On dira qu'il existe un *chemin* de s à s' dans G s'il existe une suite de sommets $s_1 \dots s_n$ telsque $s = s_1$, $s_n = s'$ et pour tout i de 1 à $n-1$ on ait (s_i, s_{i+1}) est une arête du graphe G.

La longueur du chemin de s à s' est égale à $n - 1$ (en particulier la longueur du chemin est 0 si $s = s'$).

Arbre Un arbre est un graphe G dans lequel il existe un sommet particulier r (appelé la *racine*) tel que pour tout sommet s de G il existe un chemin et un seul de r à s.

Sous-arbre Soit G un arbre et s un nœud de G. Soit S' l'ensemble des sommets s' de G tel qu'il existe un chemin de s à s'. Il est facile de montrer que l'ensemble de sommets S' et l'ensemble d'arêtes (x, y) de G avec $x \in S'$ forme un arbre de racine s qui est appelé sous-arbre de G de racine s.

7.2.2 Terminologie

Les sommets d'un arbre sont appelés des **nœuds**. À chaque nœud est associé un arbre à savoir le sous-arbre de racine ce nœud.

À chaque nœud est souvent associé une valeur (par exemple les opérateurs ou les constantes dans les expressions arithmétiques).

Un arbre traduit une organisation hiérarchique d'un ensemble de **nœuds**.

Pour un arbre donné de racine r, on distingue :

– ses **fils** : sous-arbres dont la racine est liée par une arête à r

- ses **feuilles** : les nœuds qui n'ont pas de fils
- ses **nœuds internes** : les nœuds qui ne sont pas des feuilles

Définitions associées à un nœud

père Le père d'un nœud n correspond au nœud précédant immédiatement n dans le chemin issu de la racine de l'arbre et conduisant jusqu'au nœud n

ancêtres Les ancêtres d'un nœud n correspondent à l'ensemble de tous les nœuds précédant n dans le chemin issu de la racine de l'arbre et conduisant jusqu'au nœud n

arité (ou degré) d'un nœud le nombre de ses fils

niveau Le niveau d'un nœud correspond à la longueur du chemin reliant la racine à ce nœud. La racine a pour niveau 0.

Définitions associées à un arbre

profondeur longueur maximum des chemins de la racine à une feuille

largeur nombre maximum de nœuds dans un même niveau

degré nombre maximum de fils d'un nœud de l'arbre

On distingue plusieurs types d'arbres, selon le nombre de fils des nœuds de l'arbre est fixe, borné ou pas

arbre binaire arbre dont tous les nœuds (internes) ont au plus 2 fils.
Exemple : un arbre généalogique des ascendants d'une personne est un arbre binaire.

arbre n-aire un nœud a au plus n fils

arbre quelconque le nombre de fils n'est pas borné

Arbre valué Les définitions précédentes concernent la forme de l'arbre. Dans les applications informatiques, une valeur est en générale associée à chaque sommet de l'arbre. À deux sommets différents de l'arbre peuvent être associées deux valeurs identiques. Dans la suite, on utilise le mot arbre pour désigner un arbre valué.

7.2.3 Définitions récursives

Il est possible de donner une définition récursive des arbres. Un arbre est soit une feuille, soit un nœud interne qui peut être décomposé en la racine et les fils qui sont également des arbres.

Lorsque l'on considère des arbres avec des valeurs associées, il faut de plus associer une valeur à chaque feuille et chaque nœud interne.

Arbre vide La notion d'**arbre vide** peut être utile pour uniformiser la représentation et ne pas distinguer les feuilles des nœuds internes. Un arbre vide se comporte comme une feuille à laquelle il n'est pas associé de valeur. Une feuille avec valeur peut alors se représenter comme un nœud interne dont tous les fils sont des arbres vides.

Lorqu'un arbre est non vide, il est caractérisé par sa racine à laquelle est associée une valeur et ses fils, qui sont **eux-mêmes des arbres**.

On constate donc que la définition d'un structure d'arbre est intrinsèquement récursive. Comme pour les séquences, cette récursivité structurelle va avoir un impact sur la méthode d'analyse des fonctions s'appliquant à des arbres.

Typiquement on peut s'attendre à ce que le schéma général de la réalisation d'une fonction travaillant sur un arbre contienne :

- une analyse par cas
- pour les cas où l'arbre a été construit à partir de plusieurs arbres, un appel récursif pour chaque sous-arbre utilisé dans la phase de constrution.

7.3 Arbres binaires

Les arbres binaires sont des arbres de degré 2 c'est-à-dire que tout nœud a au plus deux fils. Ils sont simples à mettre en œuvre et utilisés dans de nombreuses modélisations.

7.3.1 Première définition

Transparent

```
SPÉCIFICATION:
{Domaine}
arbre_bin {paramétré par <élément>}
{Constantes}
arbre_vide : arbre_bin
{Fonction de construction}
cons_ab (a: arbre_bin , e: <élément> , b: arbre_bin) → arbre_bin*
  {construit un arbre à partir d'une valeur pour la racine,}
  {d'un fils gauche et d'un fils droit}
{Fonctions d'accès}
valeur_rac (a : arbre_bin*) →  <élément>
fils_gauche, fils_droit (a: arbre_bin*) →  arbre_bin
  {accèdent respectivement à la racine, au fils gauche et au fils droit}
```

Fin 7.7

7.3.2 Seconde définition

Sans le concept d'arbre vide, il est nécessaire de distinguer trois opérations de construction en fonction du type de nœud (suivant s'il a 0, 1 ou 2 fils). De ce fait, il est également nécessaire d'introduire des opérations de test afin de pouvoir distinguer les différents types de nœuds.

Transparent

```
SPÉCIFICATION:
{Domaine}
arbre_bin  {paramétré par <élément>}
{Fonctions de construction}
cons_feuille (e : <élément> )→ arbre_bin
cons_unaire (e: <élément> , a: arbre_bin) → arbre_bin
cons_ab (a: arbre_bin , e: <élément> , b: arbre_bin ) → arbre_bin
{Fonctions de test}
est_feuille, est_unaire, est_binaire (a: arbre_bin ) →  bool
{Fonctions d'accès}
valeur_rac (a : arbre_bin) → <élément>
fils_gauche, fils_droit, fils_unique (a: arbre_bin) →  arbre_bin
```

Fin 7.8

7.3.3 Mise en œuvre en CAML

Définition 1

Transparent

```
type 'a arbre_bin =
          AVide
        | ABin of 'a arbre_bin * 'a * 'a arbre_bin ;;

(* Operations de construction*)
let arbre_vide = AVide;;
let cons_ab(gauche,racine,droit) = ABin(gauche,racine,droit) ;;

(* Fonctions d'acces*)
let valeur_rac a  = match a with ABin(_,racine,_) -> racine
                                 | _    -> failwith "Arbre Vide";;
let fils_gauche a = match a with ABin(gauche,_,_) -> gauche
                                 | _    -> failwith "Arbre Vide";;
let fils_droit a  = match a with ABin(_,_,droit) -> droit
                                 | _     -> failwith "Arbre Vide";;
```

Fin 7.9

Définition 2

Transparent

```
type 'a arbre_bin_2 =
          Feuille of 'a
        | AUn of 'a * 'a arbre_bin_2
        | ABin of 'a arbre_bin_2 * 'a * 'a arbre_bin_2 ;;

(* Operations de construction*)
let cons_arbre racine  = Feuille(racine) ;;
let cons_unaire_d racine arbre  = AUn(racine,arbre) ;;
let cons_ab gauche racine droit = ABin(gauche,racine,droit) ;;
```

Fin 7.10

Transparent

```
(* Operations d'accès*)
let valeur_rac a   = match a with
                         Feuille(r)  -> r
                       | AUn(r,_)   -> r
                       | ABin(_,r,_) -> r ;;
let fils_gauche a  = match a with
                         ABin(g,_,_) -> g
                       | _ -> failwith "Pas de fils gauche";;
let fils_droit a  = match a with
                         ABin(_,_,d) -> d
                       | _ -> failwith "Pas de fils droit";;
let fils_unique a  = match a with
                         Aun(_,b)    -> b
                       | _ -> failwith "Pas de fils unique";;
```

Fin 7.11

Transparent

```
(* Operations de test*)
let est_feuille a  = match a with
                         Feuille(_)  -> true
                       | _ -> false;;
let est_unaire a = match a with
                       | AUn(_,_)  -> true
                       | _ -> false;;
let est_binaire a  = match a with
                       | ABin(_,_,_)  -> true
                       | _ -> false;;
```

Fin 7.12

Transparent

Exemple : Soit à représenter l'arbre :

```
   4
  / \
 3   8
    / \
   7   9
   |
   5
```

```
# let ab = ABin(ABin(AVide,3,AVide),
                4,
                ABin(ABin(ABin(AVide,5,AVide),7,AVide),
                     8,
                     ABin(AVide,9,AVide)));;
ab: int arbre_bin = ABin(ABin(AVide,3,AVide), ...)

# let ab2 = ABin(Feuille(3),
                 4,
                 ABin(AUn(7,Feuille(5)),8, Feuille(9)));;
ab2 : int arbre_bin_2 = ...
```

Fin 7.13

7.3.4 Opérations sur les arbres binaires

Nous nous intéressons maintenant à la construction des opérations suivantes :

1. taille d'un arbre binaire
2. profondeur d'un arbre binaire
3. parcours d'un arbre binaire
4. recherche d'un élément dans un arbre binaire

Schéma général d'analyse d'une fonction sur un arbre

Transparent

Cas de la définition 1 : Un arbre binaire peut être soit un arbre vide soit non vide. On peut donc s'attendre à ce que le schéma général d'une fonction travaillant sur un arbre de cette sorte corresponde à une décomposition par cas du type :

f : (a) ↦ **si** a = arbre_vide
alors : <une valeur>
sinon : g(a, f(fils_gauche(a)), f(fils_droit(a)))

Fin 7.14

Transparent

Cas de la définition 2 : Dans ce cas il y a trois sortes d'arbres construits. Le schéma général se présente donc sous la forme d'une décomposition par cas selon le type d'arbre, à l'aide des opérations de tests. Pour chaque construction récursive, on retrouve un ou deux appels récursifs.

f : (a) ↦ **selon le cas** a
est_feuille(a) : g_1(a)
est_unaire(a): g_2(a,f(fils_unique(a)))
est_ab(a) : g_3(a,f(fils_gauche(a)),f(fils_droit(a)))

Fin 7.15

Taille d'un arbre binaire

Transparent

Problème : Construire une fonction calculant le nombre de nœuds d'un arbre binaire.

SPÉCIFICATION:
`taille_abin` (a : `arbre_bin`) → `entier`
{ renvoie le nombre de noeuds de a}

Fin 7.16

Analyse :
(suivant la définition 1) Equation récursive de la forme :

taille_abin(a) = g(a, taille_abin(fils_gauche(a)), taille_abin(fils_droit(a)))

Si l'on connaît déjà la taille du fils gauche et celle du fils droit, on peut définir la taille de l'arbre entier par :

taille_abin(a) = 1 + taille_abin(fils_gauche(a)) + taille_abin(fils_droit(a))

Ici la fonction g(x,y,z) = 1+ y + z

Remarque : Cette équation n'est correctement typée que dans le cas où l'arbre a est non vide. Dans le cas d'un arbre vide on peut prendre 0 comme taille de l'arbre.

Transparent

RÉALISATION:
taille_abin : (a) ↦ **si** a = arbre_vide **alors** : 0 **sinon** : 1 + taille_abin(fils_gauche(a)) + taille_abin(fils_droit(a))

Fin 7.17

Transparent

Codage en CAML :

```
# let rec taille_abin a = match a with
              AVide -> 0
            | ABin(fg,_,fd) -> 1 + (taille_abin fg) + (taille_abin fd) ;;
taille_abin : 'a arbre_abin -> int = <fun>

# taille_abin ab ;;
- : int = 6
```

Fin 7.18

Profondeur d'un arbre binaire

Transparent

Problème : Construire une fonction calculant la profondeur d'un arbre binaire (i.e. la longueur de la branche la plus longue).

SPÉCIFICATION:
`prof_abin` (a : `arbre_bin`) $\rightarrow$ `entier` { calcule la profondeur de a}

Fin 7.19

Analyse :
On recherche une équation récursive de la forme :

$$\text{prof\_abin(a)} = \text{g(a, prof\_abin(fils\_gauche(a)) , prof\_abin(fils\_droit(a)))}$$

Si l'on connaît déjà la profondeur du fils gauche et celle du fils droit, on peut définir la profondeur de l'arbre entier par :

$$\text{prof\_abin(a)} = 1 + \text{max(prof\_abin(fils\_gauche(a)), prof\_abin(fils\_droit(a)))}$$

Ici la fonction $g(x, y, z) = 1 + max(y, z)$

Remarque : Cette équation n'est correctement typée que dans le cas où l'arbre a est non vide. Dans le cas d'un arbre vide on peut prendre 0 comme profondeur.

Transparent

RÉALISATION:

prof_abin : (a) $\mapsto$
    si a = arbre_vide
        alors : 0
        sinon : 1 + max (prof_abin(fils_gauche(a)) , prof_abin(fils_droit(a)))

Codage en CAML :

```
# let rec prof_abin  a = match a with
              AVide -> 0
            | ABin(fg,_,fd) -> 1 + max (prof_abin fg) (prof_abin fd) ;;
prof_abin : 'a arbre_bin -> int = <fun>
# prof_abin ab ;;
    - : int = 4
```

Fin 7.20

Parcours d'un arbre binaire

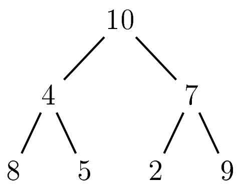
Problème : Construire une fonction calculant la séquence des valeurs d'un arbre binaire.

Remarque : la séquence des éléments peut être construite selon différents parcours :

parcours préfixe racine d'abord, puis parcours du fils gauche, puis parcours du fils droit
parcours infixe parcours du fils gauche d'abord, puis racine, puis parcours du fils droit
parcours suffixe parcours du fils gauche d'abord, puis parcours du fils droit, puis racine

Transparent

Exemple de différents parcours



parcours préfixe [10,4,8,5,7,2,9]
parcours infixe [8,4,5,10,2,7,9]
parcours suffixe [8,5,4,2,9,7,10]

Fin 7.21

Transparent

SPÉCIFICATION:
`parcours_inf` $(a : arbre\_bin\, de\ < e >) \rightarrow$ `sequence de <e>`
{ construit la séquence des éléments de a résultant d'un parcours infixe de a.}

Fin 7.22

Analyse :
On cherche une équation récursive de la forme :

parcours_inf(a) = **si** a = arbre_vide
alors : <valeur>
sinon : g(a, parcours_inf(fils_gauche(a)), parcours_inf(fils_droit(a)))

Si l'arbre est vide, il est clair que la valeur retournée doit être la séquence vide. Sinon, l'arbre contient au moins une racine et deux sous-arbres. Si l'on dispose déjà du parcours correspondant au fils gauche et de celui correspondant au fils droit il suffit de les mettre bout-à-bout en insérant la valeur de la racine.

Transparent

RÉALISATION:

parcours_inf : (a) $\mapsto$
  si a = arbre_vide
    alors : seq_vide
    sinon : concat(parcours_inf(fils_gauche(a)),
                ajout_debut(valeur_rac(a),parcours_inf(fils_droit(a))))

Fin 7.23

Transparent

Codage en CAML :

```
# let rec parcours_inf a =
      match a with
            AVide -> []
          | ABin(fg,r,fd) -> (parcours_inf fg)@(r::(parcours_inf fd));;

parcours_inf : 'a arbre_bin -> 'a list = <fun>
# parcours_inf ab;;
- : int list = [3; 4; 5; 7; 8; 9]
```

Fin 7.24

Recherche d'un élément dans un arbre

Transparent

Problème : Construire une fonction permettant de dire si un élément figure dans un arbre binaire.

SPÉCIFICATION:

`recherche_ab` (e : `<element>`; a : `arbre_bin de<element>`) $\rightarrow$ `booleen`
  { est vrai si l'élément e figure dans a}

Fin 7.25

Analyse :
Si l'arbre est vide, il est clair que la valeur retournée doit être $faux$. Dans le cas contraire on cherche à satisfaire une équation récursive de la forme :

recherche_ab(e,a) = g(e, a, recherche_ab(e,fils_gauche(a))), recherche_ab(e, fils_droit(a))

Ici, il est clair que si l'élément est présent dans le fils gauche ou droit alors il est bien dans l'arbre tout entier. Sinon, la seule possibilité est qu'il corresponde à la valeur associée à la racine de l'arbre.

Transparent

Réalisation:		
recherche_ab : (e,a) $\mapsto$	**si** a = arbre_vide	
	alors : faux	
	sinon :	e = valeur_rac(a) **ou**
		recherche_ab(e,fils_gauche(a)) **ou**
		recherche_ab(e,fils_droit(a))

Fin 7.26

Transparent

Codage en CAML :

```
# let rec recherche_ab e a =
      match a with
            AVide -> false
          | ABin(fg,r,fd) -> (e = r) or
                             (recherche_ab e fg) or
                             (recherche_ab e fd);;

recherche_ab : 'a -> 'a arbre_bin -> bool = <fun>

#recherche_ab 3 ab;;
- : bool = true
```

Fin 7.27

7.4 Arbres n-aires

Les arbres n-aires sont des généralisations du concept d'arbre binaire pour lesquelles chaque nœud interne a au plus (ou exactement) n fils. Le principe de modélisation s'adapte directement de ce qui a été présenté pour les arbres binaires. Au lieu de construire un arbre à partir d'une valeur et de deux arbres fils, on construit un arbre à partir d'une valeur et de n fils.

Comme précédemment, on peut proposer différentes définitions, en introduisant ou pas le concept d'arbre vide. Le schéma général d'analyse des fonctions sur ce type d'arbre conduit alors à des équations récursives faisant apparaître autant d'appels récursifs qu'il existe de fils à partir desquels un arbre a été construit.

En pratique, ce type de structure de données n'est cependant utilisé que pour des petites valeurs de n (e.g. 3, 4). En particulier si le nombre de fils est susceptible de beaucoup varier, on préfère généralement utiliser des structures d'arbres quelconques.

7.5 Arbres quelconques

Dans certains problèmes, il peut arriver que l'on ait besoin de construire des structures d'arbres à partir d'autres arbres sans pour autant que le nombre de fils d'un nœud soit borné. C'est le cas par exemple des arbres généalogiques. On utilise alors des structures **d'arbres quelconques**.

Dans un arbre quelconque, chaque nœud est :

– soit une feuille
– soit un nœud ayant au moins un fils

Le nombre de fils n'étant pas borné, on utilise pour regrouper les fils une structure de séquence dont les éléments sont eux-même des arbres.

7.5.1 Première définition

Transparent

SPÉCIFICATION:
{Domaine}
arbre de <e> {paramétré par <e>}
{Fonctions de construction}
cons_feuille (e: <e>) → arbre de <e> {construit un arbre réduit à une feuille}
cons_arbre (e: <e> , sa : sequence* d'arbre de <e>) → arbre de <e>
{construit un arbre à partir d'une racine d'une séquence non vide d'arbres}
{Fonctions de test}
est_feuille (a: arbre) → bool
{Fonctions d'accès}
valeur_rac (a: arbre de <e>) → <e> {valeurs de la racine d'un arbre quelconque}
fils (a : arbre de <e>) → sequence d'arbre de <e>
{séquence des fils d'un arbre quelconque non réduit à une feuille}

Fin 7.28

Remarque : Les fils étant représentés par une séquence d'arbres, on a naturellement un seul constructeur pour les nœuds de l'arbre, il n'est pas nécessaire d'introduire la notion d'arbre vide comme on l'avait fait dans le cas des arbres $n-$aires pour éviter d'avoir n sortes de nœuds.

7.5.2 Seconde définition

Une définition alternative consiste à se passer du constructeur cons_feuille et autoriser la séquence des fils à être vide, auquel cas on a une feuille.

Transparent

SPÉCIFICATION:
{Domaine}
arbre de <e> {paramétré par <e>}
{Fonction de construction}
cons_arbre (e: <e> , sa: sequence d'arbre de <e>) → arbre de <e>
{construit un arbre à partir d'une racine et d'une séquence d'arbres}
{Fonctions d'accès}
valeur_rac (a: arbre de <e>) → <e> {valeurs de la racine d'un arbre quelconque}
fils (a: arbre de <e>) → sequence d'arbre de <e>
{séquence des fils d'un arbre quelconque (éventuellement vide)}

Fin 7.29

On peut alors réaliser une fonction cons_feuille qui construit un arbre réduit à une feuille à partir d'une valeur :

RÉALISATION:
cons_feuille : $(a) \mapsto$ cons_arbre(a,seq_vide)

Transparent

Mise en œuvre en CAML : Première définition :

```
# type 'a arbre =
        Feuille of 'a
      | Noeud of 'a * 'a arbre list ;;
type arbre defined
# let a = Noeud(3, [Feuille(2);
                   Noeud(5, [Feuille(10);Feuille(15)]);
                   Feuille(8)]);;
a : int arbre = ....
```

Seconde définition :

```
# type 'a arbre =  Noeud of 'a * 'a arbre list ;;
type arbre defined
# let a = Noeud(3, [Noeud(2,[]);
                   Noeud(5, [Noeud(10,[]);Noeud(15,[])]);
                   Noeud(8,[])]);;
a : int arbre = ....
```

Fin 7.30

7.5.3 Quelques opérations sur les arbres

- calcul de la taille d'un arbre
- calcul de l'ensemble des valeurs d'un arbre
- calcul de la largeur d'un arbre

Schéma général d'analyse

Contrairement aux cas précédents, Lorsqu'on l'on veut accéder à l'ensemble des fils d'un nœud on récupère celui-ci sous la forme d'une séquence d'arbres (on parle alors de **forêt**). Il n'est donc pas possible d'appliquer directement la fonction récursivement sur chaque fils. Pour le faire, il est nécessaire d'introduire une fonction intermédiaire qui effectue ce traitement pour chaque élément de la forêt des fils.

Transparent

De ce fait, le schéma général d'analyse d'une fonction définie sur un arbre quelconque passe par la définition de deux fonctions mutuellement récursives :

f : (a) $\mapsto$ g_1(a, f_foret(fils(a)))
f_foret : (s) $\mapsto$ **si** s = seq_vide
alors : <une valeur>
sinon : g_2(s,f(premier(s)),f_foret(reste(s)))

Fin 7.31

Lorsque l'on cherche à définir une fonction f sur un arbre généralisé dont le profil est `arbre` $\rightarrow D$, on est amené à introduire une fonction f`_foret` de profil `sequence d'arbre` $\rightarrow D$ telle que la valeur de f pour un arbre puisse être calculée à partir de la valeur f`_foret` pour la forêt des fils de a et la valeur de f`_foret` pour une séquence d'arbres se calcule aisément à partir de la valeur de f pour chaque arbre de la forêt.

Calcul de la taille d'un arbre

Transparent

Problème : Calculer le nombre de nœuds d'un arbre quelconque

SPÉCIFICATION:
`taille_arbre` (a : `arbre`) $\rightarrow$ `entier`
{ calcule le nombre de noeuds dans a}

Fin 7.32

Analyse :
On cherche un schéma du type :

taille_arbre : (a) $\mapsto$ g_1(a, taille_foret(fils(a)))
taille_foret: (s) $\mapsto$ **si** s = seq_vide
alors : <une valeur>
sinon : g_s(s,taille_arbre(premier(s)),taille_foret(reste(s)))

En fait le nombre de nœuds d'un arbre quelconque correspond à la somme du nombre de nœuds de chacun de ses fils augmenté de 1 (pour la racine).
Il suffit d'introduire une fonction `taille_foret` de profil `sequence d'arbres` $\rightarrow$ `entier` qui représente la taille de la forêt définie come la somme des tailles des arbres constituant la forêt

Transparent

RÉALISATION:
taille_arbre : (a) $\mapsto$ 1 + taille_foret(fils(a))
taille_foret(s) $\mapsto$ **si** s = seq_vide
alors : 0
sinon : taille_arbre(premier(s))+taille_foret(reste(s))

Fin 7.33

Transparent

Codage en CAML :

```
let rec taille_arbre a = match a with
             Noeud(_,fils) -> 1 + (taille_foret fils)

and taille_foret s = match s with
                []  -> 0
              | a::reste -> (taille_arbre a) + (taille_foret reste);;
```

Fin 7.34

Calcul de l'ensemble des valeurs d'un arbre

Transparent

Problème : Calculer l'ensemble des valeurs associées aux nœuds d'un arbre quelconque

SPÉCIFICATION:
`liste_arbre` (a : `arbre de <e>`) → `sequence de <e>`
{ construit une séquence formée des éléments de a}

Fin 7.35

Analyse :
On cherche un schéma du type :

liste_arbre(a) = **si** est_feuille(a)
alors : g_1(a)
sinon : g_2(a, liste_foret(fils(a)))

liste_foret(s) = **si** s = seq_vide
alors : <une valeur>
sinon : g_3(s,liste_arbre(premier(fils)),liste_foret(reste(a)))

En fait, si l'on sait déjà extraire l'ensemble des tous les nœuds constituant la foret des fils, Il suffit de rajouter à cet ensemble la valeur associée à la racine de l'arbre.

L'ensemble des nœuds de la forêt des fils s'obtient en concaténant l'ensemble des nœuds du premier fils à ceux des autres fils. D'ou la réalisation :

Transparent

RÉALISATION:

liste_arbre : (a) ↦ ajout_debut(valeur_rac(a),liste_foret(fils(a)))

liste_foret : (foret) ↦ **si** foret = seq_vide
alors : seq_vide
sinon : **soit** a=premier(foret),f'=reste(foret),
concat(liste_arbre(a),liste_foret(f'))

Fin 7.36

Transparent

Codage en CAML :

```
let rec liste_arbre a = match a with
            Noeud(e,fils) -> e::(liste_foret fils)

    and  liste_foret s =
          match s with
             [] -> []
             | a::reste -> (liste_arbre a) @ (liste_foret reste);;
```

Fin 7.37

Largeur d'un arbre

Transparent

Problème : Calculer le plus grand nombre de nœuds se trouvant à un même niveau de profondeur dans un arbre.

SPÉCIFICATION:

`larg_arbre` (a : *arbre*) → `entier`
{ calcule la largeur de a}

Fin 7.38

Analyse :
L'analyse n'est pas aussi simple que précédemment car cette fois la largeur d'un arbre ne peut s'exprimer comme une simple fonction de la largeur des fils de cet arbre.

En fait il faut considérer les ensembles successifs de nœuds situés à une même profondeur. Ceci revient à comparer les longueurs des forêts constituées des nœuds situés à une même profondeur dans l'arbre initial.

On généralise la fonction de largeur d'un arbre pour calculer la largeur d'une foret d'arbres. On considère ensuite les forêts successives des nœuds de niveau 1 puis de niveau 2 etc ...

Pour chaque foret, il est possible de calculer récursivement la forêt formée des fils de chaque arbre de cette forêt. Il est ensuite facile de se convaincre que la largeur d'une forêt est le maximum du nombre d'arbres dans la forêt et de la largeur de la forêt de ses fils. On spécifie trois fonctions :

SPÉCIFICATION:

`larg_arbre` (a : `arbre`) → `entier`
{ calcule la largeur de l'arbre a}
`larg_foret` (f : `sequence d'arbres`) → `entier`
{ calcule la largeur de la forêt d'arbre f}
`foret_des_fils` (f : `sequence d'arbres`) → `sequence d'arbres`
{ calcule la forêt des fils de f}

Transparent

RÉALISATION:

larg_foret : (f) ↦
  si f = seq_vide
    alors : 0
    sinon : max(longueur(f),larg_foret(foret_des_fils(f)))
foret_des_fils : (f) ↦
  si f = seq_vide
    alors : seq_vide
    sinon : **soit** a = premier(f), f' = reste(f),
        concat(fils(a),foret_des_fils(f'))
larg_arbre(a) : (a) ↦ larg_foret(ajout_debut(a,seq_vide))

Fin 7.39

Transparent

Codage en CAML :

```
let rec foret_des_fils f = match f with
          [] -> []
        | Noeud(_,fils)::reste -> fils @ (foret_des_fils reste);;

let rec larg_foret f =
       if f = [] then 0
       else max (longueur foret) (larg_foret (foret_des_fils foret));;

let larg_arbre a = larg_foret [a] ;;
```

Fin 7.40