Langages et paradigmes

Olivier Lecarme

Master d'Informatique, première année

2007–2008

Cinquième partie V

Types

Généralités

- Même pour les langages dits « non typés » le concept de type de données est fondamental.
  - pour ces langages le type n'est pas déterminé par déclaration mais par les opérations elles-mêmes
  - l'addition entière et l'addition réelle sont distinctes
  - n'importe quelle valeur peut être considérée comme de n'importe quel type
  - l'idée est de faire du langage d'assemblage de haut niveau : PL/360, Bliss, BCPL

généralités

- Dans les langages courants, les types fournissent le contexte des opérations :
  - opérateurs surchargés
  - taille d'un objet dynamique
  - appel du constructeur de l'objet
- Ils limitent l'ensemble d'opérations applicables, fournissant ainsi une validation sémantique partielle.
- Points importants :
  - Signification et but des types.
  - Équivalence et compatibilité des types.
  - Aspects spécifiques des différents types.

Systèmes de typage

- Introduction
  - L'idée de typage est la caractéristique majeure des langages de haut niveau, ce qui les distingue des langages d'assemblage ou similaires (mentionnés plus haut).

introduction

- Un système de typage est :
  - un mécanisme de définition de types
  - un mécanisme associant ces définitions à certaines constructions du langage (quelles constructions ont un type) :
    - constantes
    - variables
    - champs d'articles
    - paramètres
    - sous-programmes
    - expressions

introduction

- Un système de typage est encore :
  - un ensemble de règles :
    - équivalence de types : quand les types de deux valeurs sont-ils identiques ?
    - compatibilité de types : quand une valeur d'un certain type peut-elle être utilisée dans un certain contexte ?
    - inférence de types : comment déterminer le type d'une expression à partir des types de ses composants ou de son contexte ?

introduction

- Dans un langage utilisant le polymorphisme, il faut distinguer le type d'une expression du type de l'objet qu'elle désigne.
- Les sous-programmes doivent avoir un type s'ils sont de première ou deuxième classe : en général c'est leur profil entier.

Vérification de type

- La vérification de type sert à assurer que le programme respecte les règles de compatibilité des types.
- Un langage est fortement typé s'il interdit l'application d'une opération à un objet qui n'est pas du type prévu (et fait respecter cette interdiction).
  - Dans un langage faiblement typé, certaines absences de compatibilité ne sont pas considérées comme des erreurs.

vérification de type

- Le typage est statique si le langage est fortement typé et que la vérification peut être faite à la compilation.
- En fait, même pour les plus statiques des langages, certaines vérifications ne peuvent pas être faites avant l'exécution.

vérification de type

- Exemples :
  - Ada est fortement typé, et le typage est presque entièrement statique.
  - Pascal a les mêmes propriétés, mais la vérification des types articles avec variante sans champ sélecteur implique une vérification complexe à l'exécution.
  - C est assez faiblement typé : unions, sous-programmes à nombre variable de paramètres, équivalence entre pointeurs et tableaux, etc. De plus, les implémentations ne font généralement pas de vérifications à l'exécution.
  - Les langages non typés laissent au programmeur l'entière responsabilité de ses actes.

vérification de type

- Le typage dynamique est une forme de retard des liaisons, présente dans les langages qui ont cette philosophie : Lisp, Scheme, Smalltalk, Icon.
  - en Smalltalk ou Icon, chaque opération vérifie à l'exécution le type de ses opérandes, et fait éventuellement les conversions nécessaires
  - en Lisp ou Scheme, la vérification peut éventuellement être court-circuitée, aux risques et périls du programmeur
- La portée dynamique implique en général le typage dynamique : le compilateur ne sait pas quel objet un nom désignera, et ne connaît donc pas son type.
- Lisp, Scheme, Smalltalk ou Icon permettent à un nom de désigner n'importe quel objet.

vérification de type

- Au contraire, le polymorphisme d'Eiffel ou Oberon est compatible avec un typage statique : les types dérivés ou étendus doivent accepter toutes les opérations du type de base.
- en ML, le compilateur effectue une inférence de type générale qui évite la déclaration, mais du coup le typage peut être statique.

Définition de type

- Les langages anciens fournissent un ensemble de types prédéfinis réduit, et rien d'autre.
- Par exemple, Algol 60 fournit les entiers, réels et booléens, et des tableaux de ces valeurs.

définition de type

- La déclaration n'existe pas toujours :
  - Fortran et Basic ne demandent même pas la déclaration des variables simples, avec une règle déterminant le type (entier ou réel) d'après la première lettre du nom.
  - Bliss et PL/360 n'ont pas de types du tout.
  - ML détermine les types par inférence.
  - Les langages à typage dynamique attendent l'exécution pour déterminer le type.
- Tous les autres langages obligent à déclarer le type de chaque objet manipulé.

définition de type

- Une déclaration introduit un nom et en détermine la nature et les caractéristiques.
- Une définition de type décrit un type particulier, et peut servir à déclarer un nom de type, ou bien à déclarer directement une variable de ce type.
- Dans certains cas on peut séparer la déclaration du type de sa définition :
  - déclaration anticipée d'un type pointeur ou d'un sous-programme
  - déclaration d'un type opaque

définition de type

- Un type peut être considéré de plusieurs manières :
  - Manière dénotationnelle : un type est un ensemble de valeurs :
    - l'ensemble de valeurs est un domaine
    - toute construction prend une valeur dans un domaine
    - une mémoire est une application des noms sur leurs valeurs
    - une fonction est une application d'une mémoire sur une autre mémoire
    - tout s'exprime en termes d'opération sur des ensembles
    - c'est la manière des spécifications sémantiques

définition de type

- Autres manières de considérer un type
  - Manière constructive : un type est soit prédéfini (ou primitif) soit composé à partir d'autres types :
    - les types prédéfinis sont en général simples : entier, réel, booléen, caractère
    - les constructeurs de types sont l'article, le tableau, l'ensemble, etc.
    - c'est la manière des langages des années 70 : Algol W, Algol 68, Pascal, Ada

définition de type

- Manière abstraite : un type est une interface formée d'un ensemble d'opérations et de leur sémantique :
  - on insiste sur la signification du type plus que sur son implémentation
  - premiers langages : Simula–67 et Smalltalk
  - point de vue des langages à objets

Classification des types

- Types simples ou scalaires
  - Booléens ou logiques :
    - souvent représentés, au moins de manière interne, par 0 et 1
    - PL/I et C n'ont pas de booléens
    - Icon non plus, mais à cause du concept général d'échec ou réussite d'une évaluation

types simples

- Caractères :
  - dans les langages anciens, pas de manipulation directe, à cause de l'architecture de mots des machines
  - introduction à partir d'Algol 68 et Pascal, avec un alphabet plus ou moins approprié
  - alphabet ASCII (norme américaine, 7 bits) pour énormément de langages, y compris développés en Europe
  - alphabet ISO-8859 (norme internationale, 8 bits) à partir des langages ayant une norme ISO
  - passage avec Java à Unicode (pas d'organisme de normalisation, 16 bits)

types simples

- Nombres :
  - certains langages ne connaissent qu'un type pour les réels
  - certains langages (C, Fortran) proposent deux longueurs différentes pour les réels mais ne précisent pas leurs relations
  - Algol 68 propose les préfixes **long** et **short** pour qualifier les réels, mais encore une fois sans spécifier les propriétés
  - Ada permet de spécifier la précision minimum, charge à l'implémentation de la respecter au moins
  - C et Modula-2 distinguent les entiers des cardinaux (entiers sans signe)

types simples

- encore des nombres :
  - Fortran et Scheme proposent des nombres complexes
  - très rarement les langages proposent des nombres rationnels
  - Scheme et Icon proposent des entiers de taille illimitée
  - Cobol et PL/I proposent des nombres décimaux
  - Ada propose des nombres en virgule fixe
  - nombres décimaux ou virgule fixe sont nécessaires pour les calculs financiers et commerciaux

types simples

- Types énumérés :
  - Introduits par Wirth pour Pascal.
  - Valeurs ordonnées.
  - Utilisation dans des boucles, comme indices de tableaux.
  - Type distinct des entiers, contrairement à C.
  - En Ada :
    - on peut spécifier le nombre ordinal
    - le même nom peut servir dans plusieurs types énumérés
    - le type caractère est un type énuméré

types simples

- Types intervalles :
  - Eux aussi introduits par Wirth pour Pascal.
  - Pris dans les types simples discrets : entiers, caractères, énumérés.
  - En Ada :
    - `type note is new integer range 0..20 ;` est un type dérivé du type `integer` par une contrainte ; il est incompatible avec son type parent
    - `subtype travail is semaine range lundi..vendredi ;` est un sous-type contraint du type `semaine`, avec lequel il est compatible

Types composites

Types composites, construits ou structurés, créés par application d'un constructeur de type à un ou plusieurs autres types.

types composites

- Articles
  - introduits par Cobol (mais pas comme de vrais types)
  - repris par PL/I et Algol W, puis à peu près tous les langages
  - collection hétérogène de champs
  - produit cartésien des types des champs

types composites

- Unions
  - introduites par Algol 68
  - restreintes par Pascal aux articles
  - reprises dans C
  - remplacées par l'extension de type d'Oberon

types composites

- Tableaux
  - le seul type composite présent partout
  - application du type des indices sur le type des composants
  - cas particulier des tableaux de caractères

types composites

- Ensembles
  - introduits par Pascal sous une forme limitée aux ensembles de types discrets
  - forme complètement générale dans SETL
  - ensemble de caractères dans Icon
  - également ensembles généraux en Icon, mais sous forme limitée

types composites

- Pointeurs
  - forme spéciale de valeur-G
  - référence à un objet du type de base
  - moyen d'implémenter des types récursifs

types composites

- Listes
  - séquence d'éléments, sans idée d'application d'un type d'indice
  - longueur variable
  - accès séquentiel
  - définition récursive
  - type fondamental dans les langages fonctionnels

types composites

- Fichiers
  - représentation des données extérieures au programme
  - peuvent être traités comme des tableaux
  - concept de position courante et accès séquentiel
  - limitations liées aux contraintes du matériel

Orthogonalité

- Facilite en principe l'apprentissage du langage.
- Les langages d'expression sont plus orthogonaux que les autres.
- Nécessité d'un type pour les constructions qui ne rendent pas de résultat : `void` en Algol 68 ou C, `null` en Icon.
- Icon procède différemment quand l'évaluation échoue.

orthogonalité

- Le système de types de Pascal est plus orthogonal que ceux qui le précèdent, mais :
  - une seule partie variante par article
  - une fonction ne peut pas rendre un résultat structuré
  - le type fichier ne permet ni affectation ni comparaison
- Puisque les types simples ont leurs constantes littérales, l'orthogonalité demande la même chose pour les types composites :
  - Pascal et Modula-2 ne le permettent pas
  - Ada le fait de la manière la plus générale, par positio ou par nom

Vérification des types

- Équivalence de types
  - Il existe deux manières différentes de définir l'équivalence de types :
    - l'équivalence structurelle s'appuie sur le contenu des définitions de types
    - c'est celle d'Algol 68, C ou ML, et celle des premières implémentations de Pascal
    - l'équivalence nominale s'appuie sur l'apparition dans le programme des définitions de types
    - c'est celle de Pascal depuis sa normalisation, des descendants de Pascal et en particulier Ada, de Java

Équivalence structurelle

- la forme de la déclaration ne compte pas :
  type `toto` = **record** `a, b : real` **end** ;
  est équivalent à :

```
type jojo = record
  a, b : real
end ;
```

- et aussi à :

```
type lili = record
  a : real ;
  b : real end ;
```

équivalence structurelle

- mais pas à :

```
type lili = record
  b : real ;
  a : real end ;
```

  ... sauf en ML.

- Le compilateur doit faire une comparaison récursive de la représentation interne des définitions de types, ce qui donne une vision de plutôt bas niveau de l'équivalence.
- Elle peut confondre des types que le programmeur voudrait voir distincts, s'il arrive par hasard qu'ils sont décrits de la même manière.

Équivalence nominale

- Deux définitions différentes correspondent à deux types différents, même si leur écriture est identique.
- Si l'on définit un alias à un type existant, y a-t-il équivalence nominale ou non ?
  - oui en Pascal ou Modula-2 :

```
type
  toto = record ... end ;
  jojo = toto ;
```

Les types `toto` et `jojo` sont équivalents.

équivalence nominale

- mais ce n'est pas toujours souhaitable :

```
type celsius = real ;
  fahrenheit = real ;
var c : celsius ;
  f : fahrenheit ;
...
  f := c ;
```

équivalence nominale

- équivalence stricte : deux noms pour le même type ne sont pas des types équivalents (type dérivé d'Ada)
- équivalence lâche : les types sont équivalents s'ils désignent finalement la même description (Pascal, sous-types d'Ada)

Conversions et forceurs

- Contextes d'attente de types donnés
  - Dans une affectation, on attend normalement un type de partie droite identique à celui de l'objet auquel se réfère la partie gauche.
  - Dans une addition, l'opérateur est surchargé, et on s'attend à deux opérandes entiers pour une addition entière, ou deux opérandes réels pour une addition réelle.
  - Dans un appel de sous-programme, on attend des paramètres effectifs du même type que les paramètres formels.

conversions et forceurs

- Si dans tous ces cas on veut que les types soient exactement ce qui est attendu, il faut utiliser très souvent une conversion explicite, obtenue par un forceur (cast).

conversions et forceurs

- Modes de conversion
  - Types structurellement équivalents alors que le langage demande l'équivalence nominale : aucun code n'est nécessaire pour effectuer la conversion.
  - Ensembles de valeurs différents, mais même représentation pour les valeurs communes (types intervalles) : le code vérifie simplement l'appartenance au type attendu (ou ne le vérifie pas !).
  - Types avec des représentations différentes, mais avec une correspondance possible entre les valeurs (entier et réel) : le code effectue réellement la conversion, peut-être à l'aide d'instructions spécialisées du processeur.

conversions et forceurs

- Exemple en Ada

```
n : integer ;
r : real ;
t : note ;
c : celsius ;
...
c := note(n) ; - vérification d'appartenance
n := integer(t) ; - aucun code
r := real(n) ; - instruction de conversion
n := integer(r) ; - conversion et vérification
n := integer(c) ; - aucun code
c := celsius(n) ; - vérification
```

conversions et forceurs

- Changement de type sans conversion
  - En programmation de bas niveau, cela peut servir à interpréter la même donnée de plusieurs manières différentes.
  - C'est indispensable pour programmer un récupérateur de mémoire, par exemple.
  - En Ada, il faut construire les fonctions correspondantes par instantiation d'une procédure générique.
  - En C, le nom du type sert aux conversions normales, mais on peut faire des conversions sans vérification en passant par des pointeurs.

Conversions automatiques

- Pour la plupart des langages, l'équivalence de type n'est pas demandée partout :
  - ce qui est demandé est la compatibilité du type de la valeur avec le type demandé
  - pour une addition, les deux types doivent être compatibles avec le type entier, ou compatibles avec le type réel
  - pour un appel de sous-programme, les types des paramètres effectifs doivent être compatibles avec les types des paramètres formels

conversions automatiques

- La compatibilité de types est définie de manière très différente suivant les langages :
  - très stricte en Ada, où elle ne doit jamais impliquer un changement de représentation
  - un peu moins stricte en Pascal, où la conversion automatique d'entier en réel est acceptée
  - assez permissive en Fortran ou C, qui permettent les conversions entre toutes les valeurs numériques
  - extrêmement permissive en Fortran 90, qui permet des conversions entre tableaux
  - permissive et extensible en C^{++}

conversions automatiques

- Si le langage permet à la fois la surcharge et les conversions automatiques, les choses se compliquent et obligent à établir des priorités (cas des opérations arithmétiques par exemple).
- Si les pointeurs peuvent être considérés comme d'un type universel :
  - ou bien on perd toute sécurité (programmation d'un récupérateur de mémoire)
  - ou bien on place dans chaque objet pointé l'indication de son type
  - c'est le cas en Java ou Eiffel, mais aussi dans les langages à typage dynamique

Inférence de type

- Une fois connu le type des opérandes, il faut déterminer le type de l'expression.
- Cas simples :
  - le résultat d'une opération est en général du type des opérandes (sauf les comparaisons)
  - le résultat d'une fonction est déclaré dans son en-tête
  - le résultat d'une affectation (s'il y a lieu) est du type de la partie droite

inférence de type

- Cas des types intervalles :
  - si `a` est de type `0..20` et `b` de type `10..20`, quel est le type de `a + b` ?
  - en Pascal, le résultat de l'opération est du type de base des opérandes, ici `integer`
  - si on affecte une variable d'un type intervalle, il faut faire une vérification
  - un compilateur soigné peut éviter une bonne partie de ces vérifications en suivant à travers le programme les contraintes respectées par les variables

inférence de type

- Cas des types composites :
  - Quel est le type d'une constante structurée ?
  - En Pascal, `'toto'` est du type
    packed array `[1..4]` **of** `char`
  - Par conséquent, on ne peut pas l'affecter à une variable d'un type plus long !
  - En revanche, `[1, 5, 12]` est d'un type ensemble non complètement défini.
  - La solution est de ramener tout ensemble au type de base, ici **set of** `[0..max]`, où `max` est défini par l'implémentation.
  - Pour les chaînes, il faut faire des extensions au système de compatibilité de type, en complétant automatiquement à la longueur nécessaire.

Articles et unions

- Les articles sont appelés structures en Algol 68, C ou C^{++}, et types en Fortran 90.
- En Cobol, un article n'est pas un type, applicable à plusieurs objets, mais un objet spécifique.
- Dans les langages à objets, l'article n'existe pas de manière spécifique, c'est un cas particulier de classe.

Syntaxe et opérations

- La définition ressemble à une suite de déclarations de variables entre deux délimiteurs spécifiques.
- Les champs sont nommés, et non pas numérotés en général.
- En Pascal et ses descendants, la référence à un champ se fait en notation pointée : `article.champ`.
- En Cobol, PL/I et Algol 68, l'ordre est inversé : `champ` **of** `article`.

syntaxe et opérations

- L'emboîtement des définitions est permis :
  - directement en Pascal et ses descendants
  - par l'intermédiaire d'un nom de type en Fortran 90
- Les références s'emboîtent comme les définitions : `article1.article2.champ`.
- En Cobol ou PL/I, on peut omettre les intermédiaires s'il n'y a pas ambiguïté : `champ` **`of`** `article1` **`of`** `article2` peut aussi s'écrire `champ` **`of`** `article2` si cela ne peut être interprété que d'une seule manière.

Représentation en mémoire

- Normalement les champs sont rangés en mémoire de manière contiguë.
- L'ordre d'énumération est donc important.
- Les contraintes d'adressage imposées par la machine peuvent faire que des zones restent non affectées :

```
type element = record
  nom : packed array [1..2] of char ;
  numeroatomique : integer ; (* 32 bits *)
  masseatomique : real ; (* 64 bits *)
  metal : boolean
end ;
```

représentation en mémoire

- En Pascal, le préfixe **packed** demande de ne pas laisser de zone non affectée (ce qui gagne 5 octets dans le cas précédent, au prix d'une grande complication des accès).
- Certains langages permettent de réordonner les champs pour gagner de la place.
- Ada permet de spécifier la position exacte de chaque champ.
- La plupart des langages permettent l'affectation globale.
- Cobol permet l'affectation par nom (articles distincts mais avec des champs communs).
- Ada permet la comparaison pour égalité.

Articles avec variante

- C'est la vision la plus courante du concept d'union, qui permet de conserver un typage assez strict :

```
type
  element = record
  nom : packed array [1..2] of char;
  numeroatomique : integer; (* 32 bits *)
  masseatomique : real; (* 64 bits *)
  metal : boolean;
    case naturel : boolean of
    true : (source : Pchaine;
    prevalence : real);
    false : (periode : real)
  end;
```

articles avec variante

- le champ `naturel` est le sélecteur ou le discriminant
- chaque groupe de champs entre parenthèses est une variante
- seule l'une des deux peut exister à tout moment, suivant la valeur du champ sélecteur
- les variantes partagent donc le même emplacement de mémoire, et d'ailleurs dans le cas présent ne sont pas de même taille
- la taille totale de l'article dépend donc de la valeur du sélecteur

articles avec variante

- L'idée la plus ancienne est celle de la déclaration d'équivalence de Fortran : `EQUIVALENCE (A, B, C)` indique que les trois variables mentionnées partagent le même emplacement.
- La contribution de Pascal, réutilisée en Modula-2 et Ada, est d'intégrer l'idée dans celle d'article et de fournir les moyens de faire vérifier la validité des références.
- Les unions de C nécessitent une notation plus compliquée.

articles avec variante

- Les problèmes principaux sont la sécurité d'accès :
  - nulle en Fortran ou C
  - totale en Algol 68 avec la nécessité de passer par un énoncé **case** spécifique
  - bonne en Pascal si le compilateur fait les vérifications nécessaires
  - cependant, la désynchronisation entre les changements de valeur du sélecteur et l'affectation aux champs n'est pas vérifiable
  - quand le champ sélecteur est omis, la sécurité nécessite que le compilateur en gère un qui est caché

Tableaux

- C'est le seul type composite qu'on trouve partout.
- Normalement homogène, sauf dans les langages à typage dynamique.
- Application du type d'indice sur le type de composant.
- Tableaux associatifs dans certains langages (Perl, Icon, C^{++}), pas considérés ici.

Syntaxe et opérations

- Référence avec le nom du tableau et une valeur d'indice.
- Utilisation de crochets en Pascal, C ou Icon.
- Utilisation de parenthèses en Fortran ou Ada :
  - en Fortran, pour limiter le jeu de caractères utilisé
  - en Ada, pour respecter le principe de référence uniforme, qui masque à l'utilisateur d'un module la manière d'implémenter les opérations

syntaxe et opérations

- Déclaration : méthodes très variées.
  - en C : `char majuscule[26] ;`
  - en Fortran : `character(26) majuscule`
  - en Pascal : `majuscule : array ['a'..'z'] of char ;`
  - en Ada : `majuscule : array (character range 'a'..'z') of character ;`

syntaxe et opérations

- Tableaux à plusieurs dimensions :
  - pas toujours considérés de la même manière
  - `matrice : array(1..10,1..10) of real ;` en Ada
  - `matrice : array [1..10] of array [1..10] of real ;` en Pascal, mais l'abréviation `] of array [` $\Rightarrow$ est permise
  - `float matrix [10][10]` en C
  - la référence n'utilisant qu'un indice est permise en Pascal ou C, elle ne l'est en Ada qu'avec une déclaration convenable

syntaxe et opérations

- Sections :
  - Une section est une partie rectangulaire d'un tableau.
  - On peut toujours se référer à un composant, pour l'examiner ou le modifier.
  - Pascal et ses descendants permettent de se référer à une ligne d'un tableau à deux dimensions, mais pas à une colonne.
  - Algol 68 et Fortran 90 permettent de se référer à un sous-tableau quelconque.
  - APL élargit toutes les opérations possibles aux tableaux, quel que soit le nombre de dimensions.

Dimensions et bornes

- Le point majeur est de savoir quand les valeurs des bornes sont fixées :
  - durée de vie globale, valeurs des bornes statiques : le compilateur alloue l'espace de manière statique (Fortran)
  - durée de vie locale, valeurs des bornes statiques : le compilateur alloue l'espace dans la zone d'activation locale (Pascal)
  - durée de vie locale, valeurs des bornes connues à l'exécution de la déclaration : le compilateur alloue l'espace dans la pile, en le faisant pointer indirectement depuis la zone d'activation locale (Algol 60, Ada)

dimensions et bornes

- autre manière de fixer les bornes :
  - durée de vie arbitraire, valeurs des bornes connues à l'exécution de la déclaration : allocation explicite dans le tas (Java)
  - durée de vie arbitraire, valeurs des bornes variables : allocation dans le tas, et création de nouveaux objets quand les valeurs des bornes changent (APL, Icon, Perl)

dimensions et bornes

- Le cas des paramètres de sous-programme peut être différent :
  - en Pascal, les paramètres tableaux conformants permettent une spécification incomplète :

```
function ProduitScalaire
  (a, b : array [inf..sup : integer] of real) : real ;
var i : integer ; res : real ;
begin
  res := 0 ;
  for i := inf to sup do res := res + a[i] * b[i] ;
  ProduitScalaire := res
end ;
```

  - idée similaire en Ada : **array** `(integer` **range** `<>)` **of** `real`

dimensions et bornes

- Chaînes de caractères :
  - parfois traitées exactement comme des tableaux, si le langage fournit le nécessaire : Icon, Perl
  - mécanisme similaire mais description différente en Java
  - extensions spécifiques : Pascal, Ada

Organisation en mémoire

- Le plus souvent, rangement contigu des composants.
- Si les composants ont des contraintes d'alignement, il peut rester des trous entre eux.
- En Pascal on peut spécifier l'attribut **packed** pour éviter ces trous.

organisation en mémoire

- Pour un tableau à deux dimensions, deux manières de ranger les éléments :
  - par ligne : `t[1,1]` est suivi par `t[1,2]` (la plupart des langages)
  - par colonne : `t[1,1]` est suivi par `t[2,1]` (Fortran)
  - très significatif pour les performances des parcours répétitifs, pour utiliser les possibilités des mémoires caches

organisation en mémoire

- On peut aussi représenter le tableau à deux dimensions comme un tableau de pointeurs sur des tableaux à une dimension :
  - place supplémentaire prise par les pointeurs
  - accès plus rapide car sans multiplication
  - possibilité de lignes de longueurs variables

organisation en mémoire

- Pour un paramètre formel, la représentation nécessite un descriptif :
  - borne inférieure dans chaque dimension
  - nombre d'éléments dans chaque dimension
  - pointeur sur le tableau dans la pile
  - borne supérieure dans chaque dimension si l'on vérifie la validité des indices

Pointeurs et types récursifs

- Généralités
  - Type récursif : défini à partir de lui-même, ou contenant des références à lui-même.
  - Permet de construire des structures dynamiques telles que listes et arbres.
  - Dans les langages utilisant un modèle de référence pour les variables (Lisp, Scheme, Icon, Java pour les objets), toute variable est une référence, donc un article de type `toto` contient facilement une référence à un article de même type.

généralités

- Dans les langages utilisant un modèle de valeur (PL/I, Algol 68, Pascal, Modula-2, C, Ada), il faut le concept de pointeur, qui est une valeur servant de référence.
- En Pascal ou Modula-2, les pointeurs ne peuvent désigner que des objets dans le tas.
- Les autres langages fournissent un opérateur qui donne un pointeur sur un objet ordinaire.
- Un pointeur est plus qu'une adresse, c'est un concept de haut niveau (sauf en C !).
- S'il y a allocation dans le tas, il faut soit une désallocation explicite, soit un récupérateur de mémoire.

Syntaxe et opérations

- Les opérations sont avant tout l'allocation et la désallocation des objets pointés, le dérepérage d'un pointeur, et l'affectation d'un pointeur.

syntaxe et opérations

- L'affectation dépend du modèle utilisé :
  - Dans un langage fonctionnel, en général on utilise un modèle de référence, et l'allocation est automatique.
  - Dans la plupart des langages impératifs, on utilise un modèle de valeur (Pascal, C, Modula-2, Ada) : `a := b` n'a pour effet de faire désigner par `a` l'objet désigné par `b` que si `a` et `b` sont des pointeurs.
  - Dans les langages utilisant le modèle de référence, on s'arrange cependant pour ne pas utiliser de référence pour les objets immuables, en particulier les constantes.

syntaxe et opérations

- En Java, les deux modèles coexistent de manière explicite : `a := b` place la valeur de `b` dans `a` si les deux variables sont d'un type prédéfini, sinon c'est une affectation de pointeur.

Modèle de référence

- Modèle de référence
  - Lisp permet de construire une structure explicitement à partir de l'opération `cons`, qui construit une paire pointée : couple de pointeurs vers d'autres objets.
  - En Icon, avec le type d'article `record noeud(contenu, gauche, droite)` on peut construire un arbre :

```
racine := noeud("A", &null, &null)
racine.gauche := noeud("B", &null, &null)
racine.droite := noeud("C", &null, &null)
```

modèle de référence

- Dans un langage purement fonctionnel, la structure construite est forcément acyclique.
- En Icon c'est facile de la rendre cyclique :

```
racine.gauche.gauche := racine
```

Modèle de valeur

- Exemple en Pascal :

```
type P_noeud = ^noeud;
  noeud = record
    contenu : string;
    gauche, droite := P_noeud
  end;
```

- Définitions similaires en Modula-2, Ada ou C.

Construction

- Pas de constante pour les structures dynamiques, qui doivent être construites explicitement :

```
var arbre : P_noeud ;
begin
  new(arbre) ;
  with arbre^ do
  begin
    contenu := 'toto' ;
    new(gauche) ;
    droite := nil ;
    gauche^.contenu := 'jojo' ;
    ...
  end ;
```

construction

- En C^{++} ou Java, l'allocation de l'objet appelle de plus (s'il existe) le constructeur de l'objet, qui en initialise les composants.

Dérepérage

- Flèche verticale (représentée par un circonflexe) en Pascal ou Modula-2.
- Deux notations distinctes en C :

```
(*arbre).contenu = "toto";
arbre->contenu = "toto";
```

- Pas d'opérateur explicite en Ada :
  `arbre.contenu := 'toto'.`
- Pas non plus de dérepérage explicite en Lisp ou Icon.

Pointeurs et tableaux en C

- les tableaux de C sont interchangeables avec les pointeurs
- après les déclarations suivantes :

```
int n ;
int *p ;
int t[10] ;
```

les affectations suivantes sont possibles :

```
p = t ;
n = p[3] ;
n = *(p + 3) ;
n = t[3] ;
n = *(t + 3) ;
```

pointeurs et tableaux en C

- les programmeurs ont pris l'habitude de croire que l'arithmétique sur les pointeurs est plus efficace pour parcourir un tableau
- la lecture des définitions de types impliquant pointeurs et tableaux est particulièrement difficile : voir la différence entre `int *t[n]` et `int (*t)[n]`
- deux représentations possibles pour les tableaux à deux dimensions : tableau de tableaux ou tableau de pointeurs sur tableaux
- beaucoup d'opérations de très bas niveau pour l'accès aux tableaux compliqués, et manque total de sécurité

Références pendantes

- Il y a trois classes d'allocation pour les objets : statique, en pile ou en tas.
- Les objets en pile sont désalloués automatiquement en fin de sous-programme.
- Pour les objets en tas :
  - en PL/I, Pascal, Modula-2, C ou C^{++}, il faut une opération explicite (`dispose`, `free`, `delete`, etc.)
  - en Lisp, Ada, Icon ou Java, un objet n'est désalloué (automatiquement) que quand il n'est plus accessible

références pendantes

- Une référence pendante est un pointeur qui désigne un objet qui n'existe plus :
  - objet en tas désalloué explicitement
  - objet en pile dont le sous-programme est terminé
- En Algol 68, une variable pointeur ne peut désigner une variable de durée de vie plus courte (mais cela nécessite des vérifications dynamiques en cas de passe comme paramètre)
- En Ada, la variable pointée ne peut pas avoir une durée de vie plus courte que celle du type pointeur, ce qui peut être vérifié dynamiquement.
- On peut faire une assez bonne vérification des références en associant une clé aléatoire à tout objet alloué, et en plaçant cette clé dans la représentation du pointeur.

Autres types

- Chaînes de caractères
  - Dans certains langages, simple tableau de caractères.
  - La plupart du temps, nécessité d'un peu plus de flexibilité :
    - nécessité pour la plupart des applications
    - seule manière de permettre la concaténation
    - coût faible à cause de la structure linéaire et sans référence interne

chaînes de caractères

- Parfois distinction entre la représentation d'une constante de type caractère et une chaîne de longueur 1 (C).
- Conventions variées pour inclure dans la chaîne le délimiteur ou des caractères non imprimables.
- Implémentation par un tableau de caractères avec indication de longueur ou de fin en Pascal, C ou Ada.
- Implémentation par un bloc en tas en Lisp, Icon ou Java.
- Ensemble d'opérations très élaboré en Snobol4, Icon ou Perl.

Ensembles

- Vision la plus courante : celle de Pascal.
  - collection non ordonnée de valeurs d'un type de base
  - pour une représentation efficace, le type de base est discret et limité
  - l'implémentation est par la fonction caractéristique de l'ensemble (un bit par élément possible)
  - opérations d'union, intersection, différence et appartenance

ensembles

- Vision plus élaborée : Icon.
  - les ensembles de caractères permettent de spécifier les opérations d'analyse de chaînes
  - les ensembles sont des collections d'objets quelconques
  - sur ces derniers, opérations ordinaires et implémentation par table associative
- Langage de manipulation d'ensembles : SETL.
- Au total, type peu représenté dans les langages, et c'est dommage.

Listes

- Définition récursive comme une paire formée d'un objet et d'une liste, ou le vide.
- En Lisp, tout est liste, y compris le programme (propriété d'homoïconicité).
- Dans tout langage avec des pointeurs, le programmeur peut construire des listes.
- Sous-programmes ou classes de bibliothèque dans certains langages.

listes

- Icon appelle liste un objet qui est à la fois une liste, une file, une pile et un tableau :
  - références indicées, à partir de la gauche ou de la droite
  - possibilité de sous-tableau
  - fonctions d'adjonction et de suppression aux deux extrémités

listes

- Notation en Lisp :
  - paire pointée : `(a . b)`
  - liste : `(a b c)` est équivalent à `(a . (b . (c . nil)))`
  - correspondante immédiate avec l'implémentation
  - `(cons 'a 'b)` produit la paire pointée `(a . b)`
  - `car` fournit le premier élément d'une paire, et `cdr` le deuxième