Mais que (se) cache(-t-il derrière) ce type ?

Yann Régis-Gianas

PPS (Université Paris Diderot) – πr^2 (INRIA)

25 mars 2009

Plan

1. Bestiaire de quelques systèmes de type pour la programmation
2. Un monstre nommé GADT lâché sur un automate à pile innocent
3. Retour vers le futur (épisode 1 : les années 60)

Un type, vous dites ?

- Un *type* spécifie la forme du résultat d'un calcul avant que celui-ci n'ait lieu.
- Il représente une propriété *statique* du calcul, c'est-à-dire un *invariant*.
- Une expression est bien typée
  si elle promet de faire "bon usage" de l'évaluation de ses sous-expressions.

Quelques expressions plus ou moins bien typées

- Si e et f s'évaluent en des entiers, alors "e + f" est bien typée.
- Si e s'évalue en un entier, alors "e = "foo"" est mal typée.
- "**if true then** 42 **else** "foo"" est mal typée.
  (mais pourtant, son évaluation n'est pas problématique !)
- Si e s'évalue en un flottant, "sqrt e" est bien typée ...
  (mais si e s'évalue en un flottant négatif, alors le programme explose ! ?)

$\Rightarrow$ Que signifie "faire bon usage" d'une valeur ?

Définition formelle d'un langage de programmation

- Pour un langage de programmation, on spécifie une relation

$$e \longrightarrow e'$$

  qui signifie que l'expression e s'évalue en un pas en l'expression e'.
- Par exemple, pour les expressions arithmétiques :

$$1 + 2 + 3 \longrightarrow 3 + 3 \longrightarrow 6$$

- Certaines expressions sont bloquées :

$$1 + \text{"}foo\text{"} \not\longrightarrow$$

- Éviter ces configurations garantit la *sûreté* de l'exécution.

Définition formelle d'un système de typage

- On écrit "$E \vdash e : T$" avec :
  - E un environnement de typage (une fonction des identifiants vers les types).
  - e une expression du langage de programmation.
  - T un type.

  pour exprimer le *jugement* "être bien typé".
- On définit ce jugement à l'aide d'un ensemble fini de règles de la forme :

$$\frac{}{E \vdash 42 : \textbf{int}} \qquad \frac{E \vdash e : \textbf{int} \qquad E \vdash f : \textbf{int}}{E \vdash e + f : \textbf{int}}$$

À prouver pour chaque langage de programmation !

- Si $E \vdash e : T$ alors l'évaluation de e ne peut pas bloquer.
- Peut se prouver à l'aide de deux propriétés :
  - Préservation du typage par l'évaluation.

    $$\text{Si } E \vdash e : T \text{ et } e \longrightarrow e' \text{ alors } E \vdash e' : T$$

  - Progression de l'évaluation des expressions bien typées.

    $$\text{Si } E \vdash e : T \text{ alors}$$

    - soit e est le résultat du calcul (une valeur)
    - soit e peut s'évaluer.
    - (e n'est jamais bloquée)

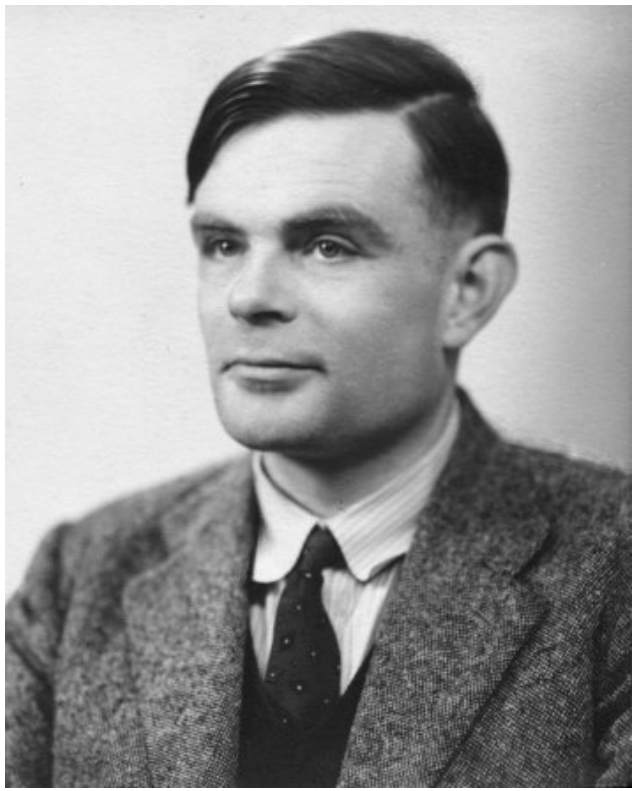
À prouver pour chaque langage de programmation !

- Si $E \vdash e : T$ alors l'évaluation de e ne peut pas bloquer.
- Peut se prouver à l'aide de deux propriétés :
  - Préservation du typage par l'évaluation.

    Si $E \vdash e : T$ et $e \longrightarrow e'$ alors $E \vdash e' : T$
  - Progression de l'évaluation des expressions bien typées.

    Si $E \vdash e : T$ alors
    - soit e est le résultat du calcul (une valeur)
    - soit e peut s'évaluer.
    - (e n'est jamais bloquée)

La Recherche sur le typage



- On *ne peut pas* décider toute propriété statiquement.

La tension

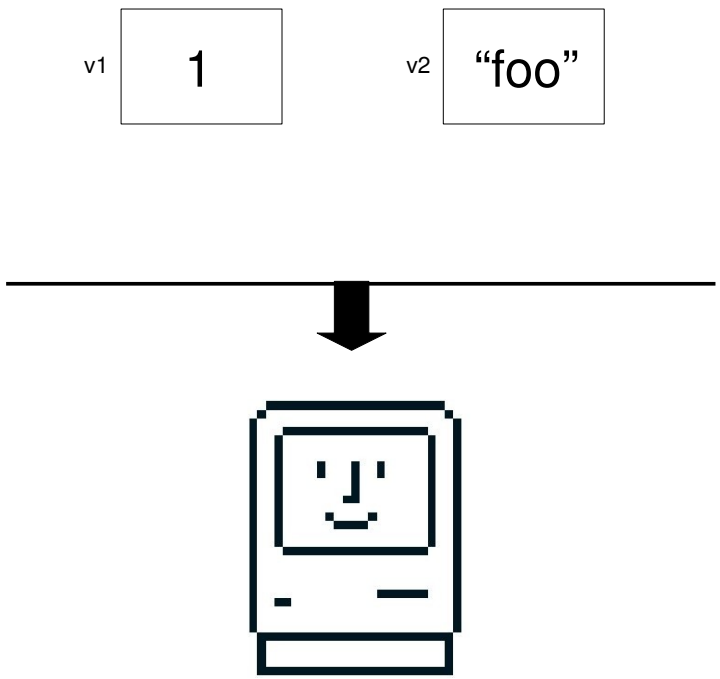
Généralité et Richesse des programmes
versus
Précision du typage

- Que faire de la racine carrée ?
  - Ne pas l'inclure dans le langage de programmation, elle est trop dangereuse !
  - Se donner un type pour les flottants positifs.
  - Lancer une exception au moment de l'exécution.
- "**fun** x $\longrightarrow$ x" a le type "int $\longrightarrow$ int" mais aussi "bool $\longrightarrow$ bool". Lequel choisir ?
- Si "1 + 1.5" et "1 + 1" sont des expressions bien typées, alors quelles sémantiques et quels types leur donner ?

Cacher pour protéger

Qui cherche l'infini n'a qu'à fermer les yeux. – Milan Kundera

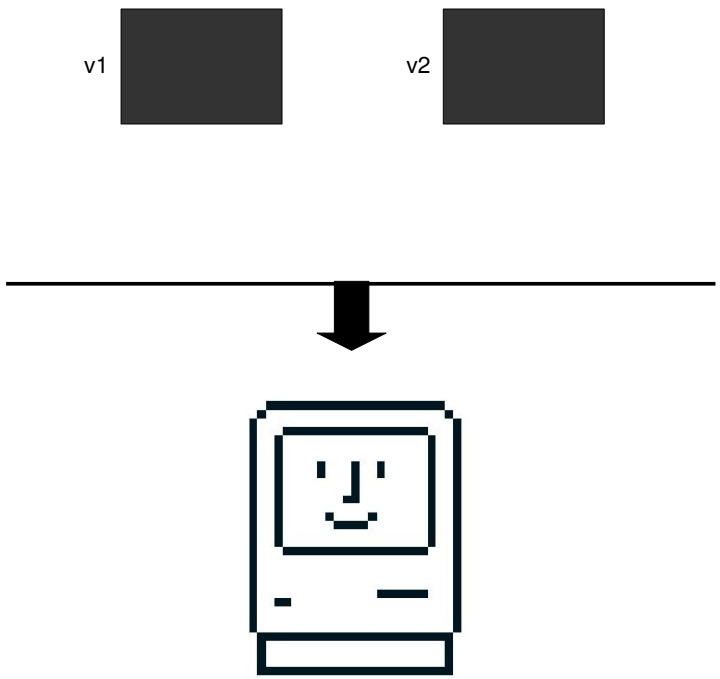
Cacher pour protéger : le polymorphisme paramétrique



- Entrées : tout type de valeur
- Promesse du programme P :

  “Je n’observerai pas les valeurs.”

Cacher pour protéger : le polymorphisme paramétrique

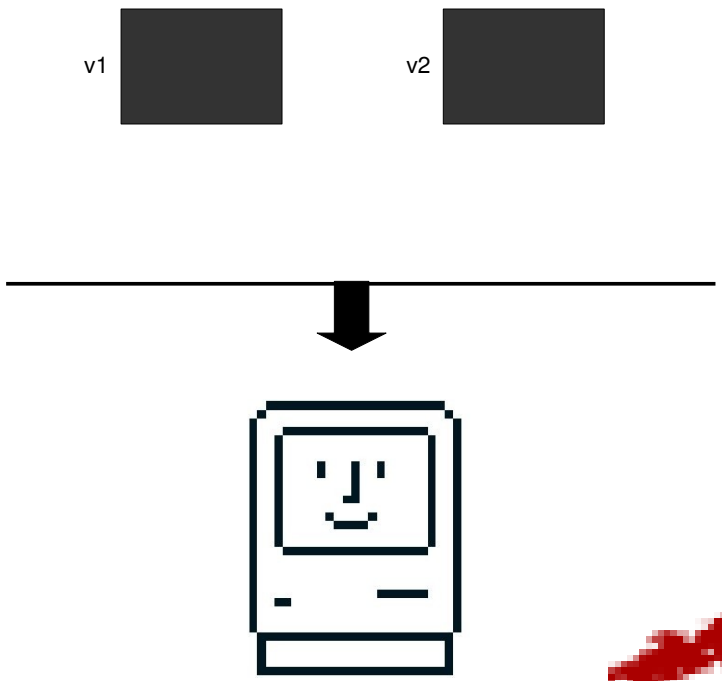


- Entrées : tout type α
- Type du programme P :

$$\forall \alpha . \alpha \rightarrow \tau$$

- $P(1)$ est valide (on choisit $\alpha = \textbf{int}$)
- $P(\text{"}foo\text{"})$ est valide (on choisit $\alpha = \textbf{string}$)

Cacher pour protéger : le polymorphisme paramétrique



- Entrées : tout type α
- Type du programme P :

$$\forall \alpha . \alpha \to \tau$$

- $P(1)$ est valide (on choisit $\alpha =$ **int**)
- $P($"foo"$)$ est valide (on choisit $\alpha =$ **string**)

Exemples de fonction polymorphe

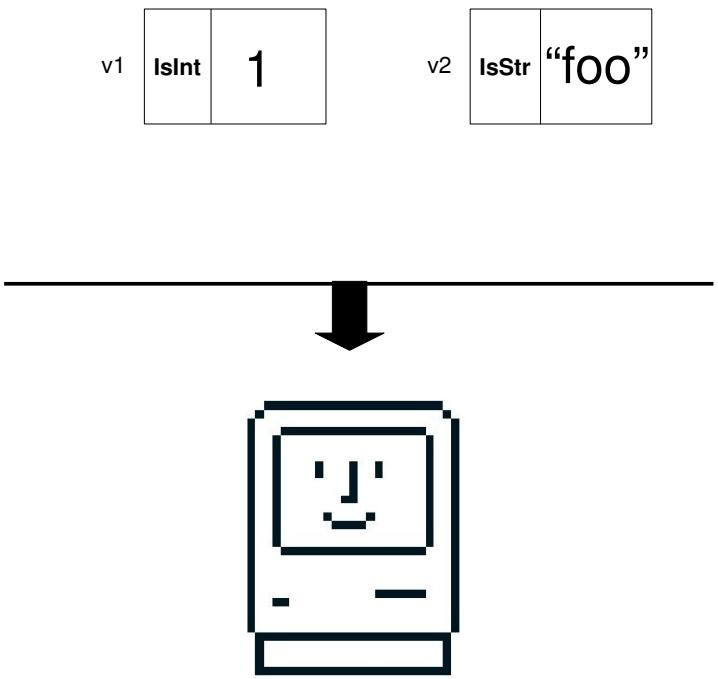
- L'identité polymorphe :

```
let id x = x
(* id : ∀α.α → α*)
```

- La construction d'une paire :

```
let mk_pair x y = (x, y)
(* mk_pair : ∀αβ.α → β → α × β *)
```

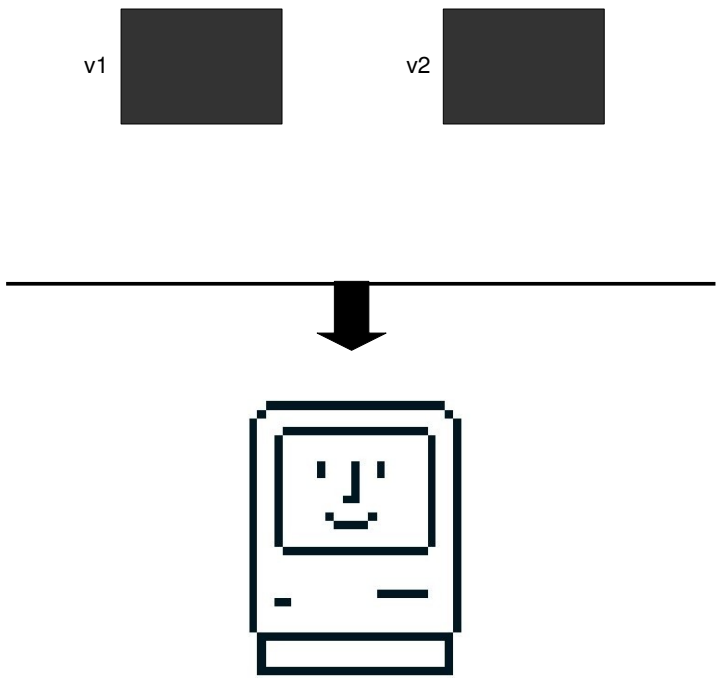
Ouvrir à qui a la clé



- Entrées : un ensemble fini de types de valeur
- Promesse du programme P :

  “Je traiterai toutes les formes de valeurs de type ϵ.”

Ouvrir à qui a la clé



- Entrées : un type *algébrique* ϵ dont les valeurs sont construites uniquement à l'aide de :

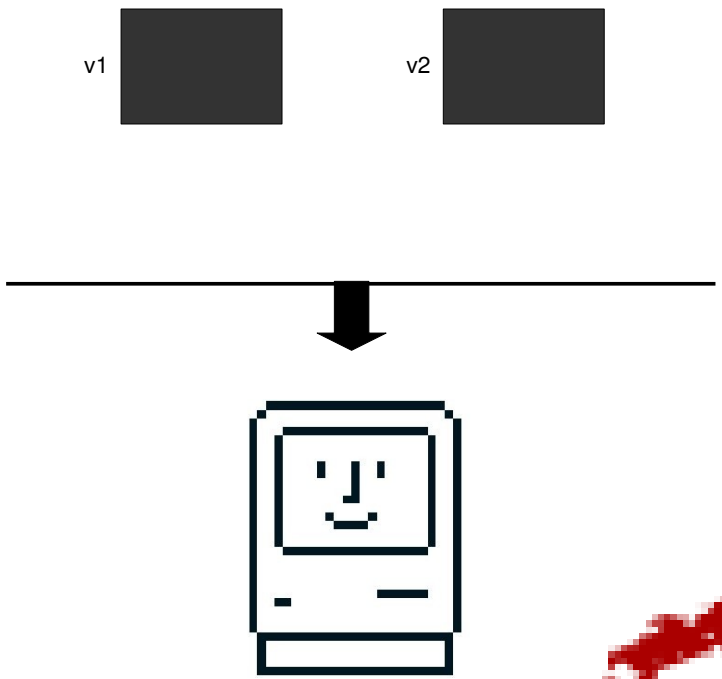
  IsInt : int $\longrightarrow \epsilon$
  IsStr : string $\longrightarrow \epsilon$

- Type du programme P :

  $$\epsilon \longrightarrow \tau$$

- $P(\mathit{IsInt}\ 0)$ est valide.
- $P(\mathit{IsStr}\ \text{"}\mathit{foo}\text{"})$ est valide.

Ouvrir à qui a la clé



- Entrées : un type *algébrique* ϵ dont les valeurs sont construites uniquement à l'aide de :

  IsInt : int $\longrightarrow \epsilon$
  IsStr : string $\longrightarrow \epsilon$

- Type du programme P :

  $\epsilon \longrightarrow \tau$

- $P(IsInt\ 0)$ est valide
- $P(IsStr\ "foo")$ est valide

Exemples de programme utilisant l'analyse de motifs

- La longueur d'une liste :

```
let rec length l =
  match l with
  | [] → 0
  | x :: xs → 1 + length xs
```

- Un interprète pour un langage jouet :

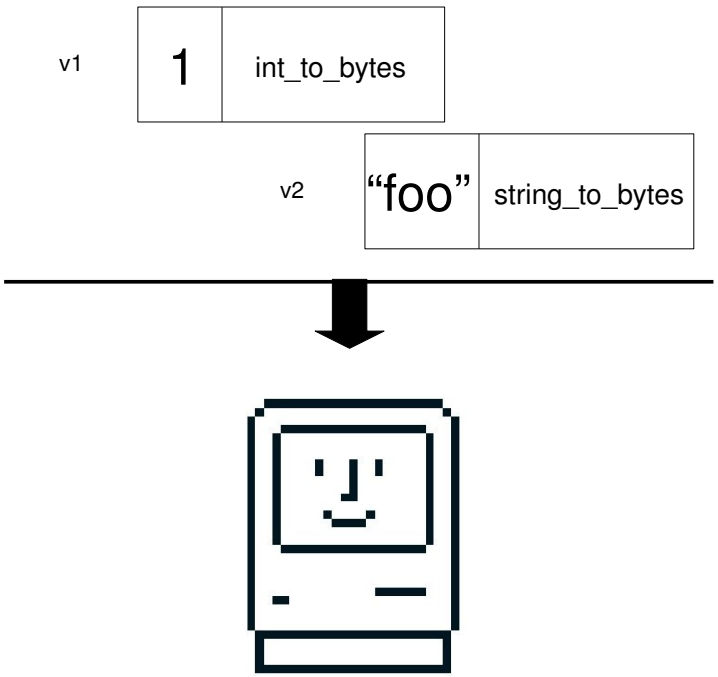
```
type term =
| Int : int → term
| Succ : term → term
| Pair : term × term → term
| Fst : term → term
| Snd : term → term
```

```
let rec eval (t : term) : term =
  match t with
  | Int i → Int i
  | Succ t → (match eval t with
              | Int i → i + 1
              | _ → assert false)
  | Pair (t1, t2) → Pair (eval t1, eval t2)
  | Fst t → (match eval t with
             | Pair (t, _) → t
             | _ → assert false)
  | Snd t → (match eval t with
             | Pair (_, t) → t
             | _ → assert false)
```

À retenir sur l'analyse de motifs

Un *test dynamique* sur l'étiquette
fournit une *information statique* concernant la donnée sur laquelle elle porte.

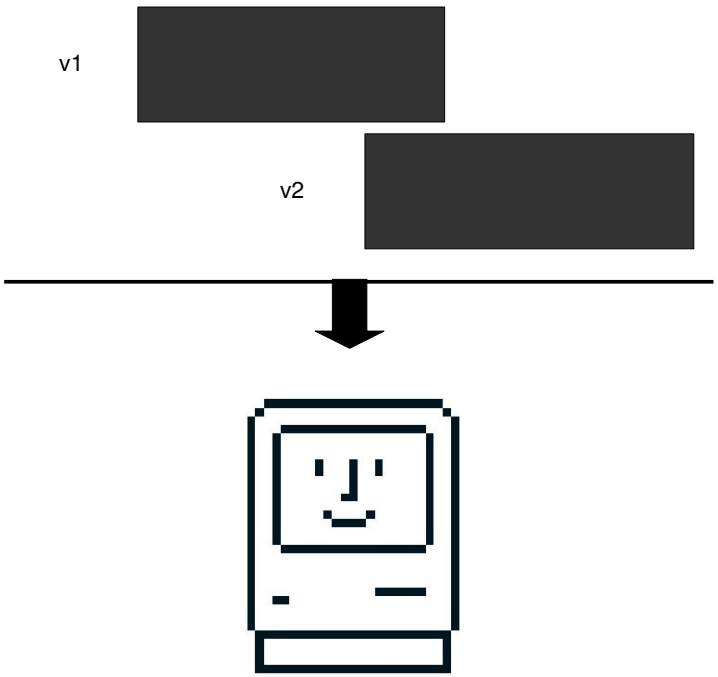
Cacher jusqu'au type



- Entrées : des couples formés d'une valeur et d'une fonction pour sérialiser cette valeur.
- Promesse du programme P :

> “Pour toute paire (v, s), je supposerai uniquement l'existence d'un type inconnu β tel que $v : \beta$ et $f : \beta \longrightarrow$ **bytes**”

Cacher jusqu'au type

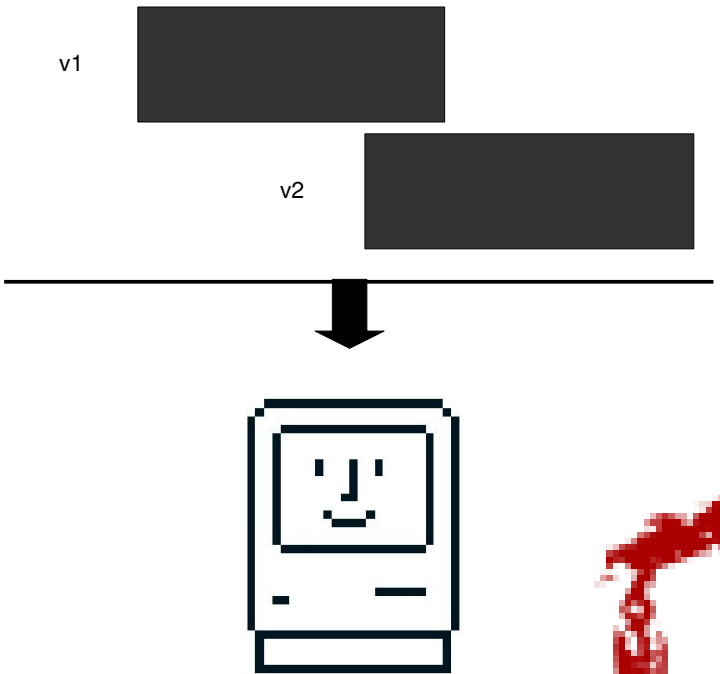


- Entrées : un type *existentiel* ϵ tel que toute valeur de ce type est produite par application du constructeur :
  $Pack : \forall\alpha.\alpha \times (\alpha \rightarrow \textbf{bytes}) \rightarrow \epsilon$
- Type du programme P :

$$\epsilon \rightarrow \tau$$

- $P(Pack(42, int_t o_b ytes))$ est valide.
- $P(Pack(\text{"}foo\text{"}, string_t o_b ytes))$ est valide.

Cacher jusqu'au type



- Entrées : un type *existentiel* ϵ tel que toute valeur de ce type est produite par application du constructeur :
  Pack : $\forall\alpha.\alpha \times (\alpha \rightarrow \textbf{bytes}) \rightarrow \epsilon$
- Type du programme *P* :
  $\epsilon \rightarrow \tau$
- *P*(*Pack*(42, *int*, ...)) est valide.
- *P*(*Pack*("coucou", "coucou".*bytes*, ...)) est valide.

Quelques exemples d'utilisation type existentiel

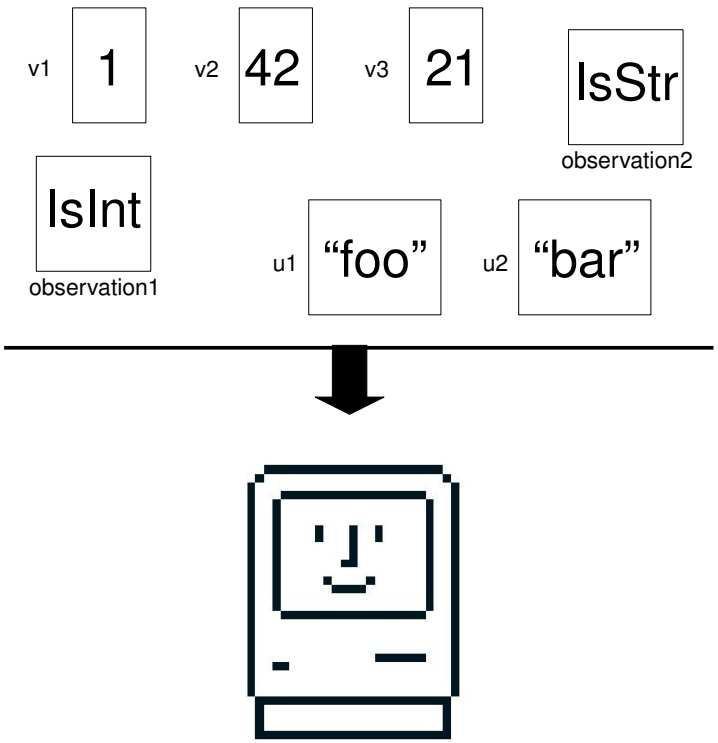
- Les types existentiels permettent l'*encapsulation*, c'est-à-dire la non divulguation des détails d'implémentation.

```
type int_stack = PackStack : forall α. {
  empty : α;
  push : int → α → α;
  pop : α → int × α
}

let m1 : int_stack = PackStack {
  empty = [];
  push = fun x xs → x :: xs
  pop = function [] → assert false | x :: xs → (x, xs)
}

let m2 : int_stack = PackStack {
  empty = (0, Array.create max_stack_size);
  push = fun x (offset, tab) →
                assert (offset < max_stack_size);
                tab.(offset) ← x; (offset + 1, tab);
  pop = fun (offset, tab) →
                assert (offset > 1);
                (tab.(offset), (offset - 1, tab));
}
```

À retenir sur les types existentiels

On peut se souvenir que des *types sont égaux*
tout en oubliant *l'exacte définition* de ces types.

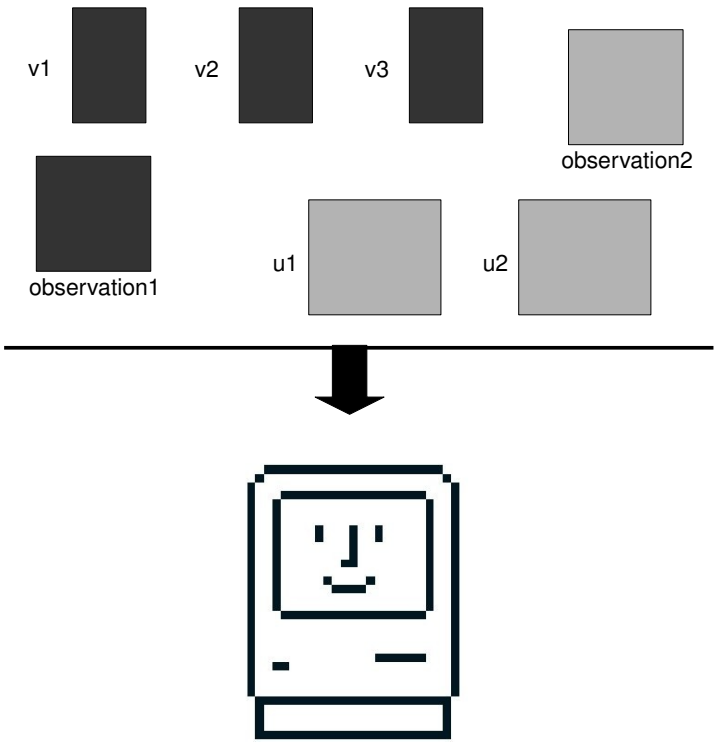
Une clé pour les types ?



- Entrées : Des valeurs et des “observations” de leur type.
- Promesse du programme P :

  “Je ne ferai pas d’autres hypothèses sur les types que celles fournies par les observations.”

Une clé pour les types ?

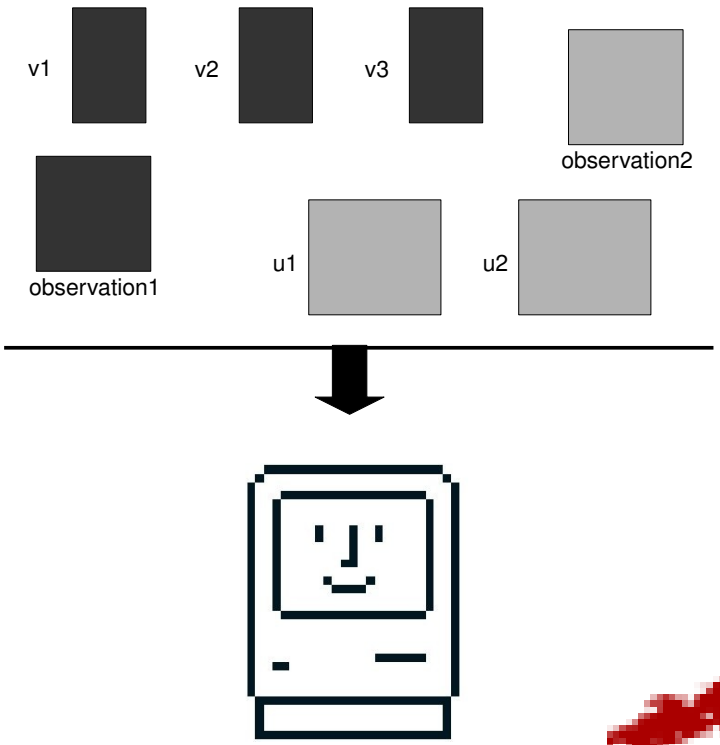


- Entrées : Des valeurs de type α et des observations de type *algébrique généralisé* $\mathrm{observation}\,\alpha$ produites par application des constructeurs :
  - $\mathrm{IsInt} : \forall\alpha[\alpha = \textbf{int}].\mathrm{observation}\,\alpha$
  - $\mathrm{IsStr} : \forall\alpha[\alpha = \textbf{string}].\mathrm{observation}\,\alpha$
- Type du programme P :

$$\forall\alpha.\mathrm{list}\,\alpha \to \mathrm{observation}\,\alpha \to \tau$$

- $P(\mathit{IsInt}, [42; 21; 1])$ est valide.
- $P(\mathit{IsStr}, [\text{"}\mathit{foo}\text{"}; \text{"}\mathit{bar}\text{"}])$ est valide.

Une clé pour les types ?



- Entrées : Des valeurs de type α et des observations de type *algébrique généralisé* $\mathrm{observation}\,\alpha$ produites par application des constructeurs :
  - $\mathrm{IsInt} : \forall\alpha[\alpha = \textbf{int}].\mathrm{observation}\,\alpha$
  - $\mathrm{IsStr} : \forall\alpha[\alpha = \textbf{string}].\mathrm{observation}\,\alpha$
- Type du programme P :
  - $\forall\alpha.\mathrm{list}\,\alpha \to \mathrm{observation}\,\alpha \to \tau$
  - $P(\mathit{IsInt}, [42; 21; \ldots])$ est valide
  - $P(\mathit{IsStr}, [\text{"foo"}; \text{"bar"}])$ est valide

Exemples de type algébrique généralisé (1/2)

- On peut écrire des programmes dirigés par les types :

```
type ty α =
| IsInt  :  forall α [ α = int ]. ty α
| IsStr  :  forall α [ α = string ]. ty α
| IsPair :  forall α 'b 'c [ α = 'b × 'c ].
    ty 'b × ty 'c → ty α
```

```
let rec serialize (desc : ty α) (v : α) : bytes =
  match desc with
  | IsInt → (* α = int *) int_to_bytes v
  | IsStr → (* α = string *) int_to_bytes v
  | IsPair (desc1, desc2) →
    (* ∃βγ.α = β ∗ γ *)
    pair_to_bytes (serialize (fst v)) (serialize (snd v))
```

Exemples de type algébrique généralisé (2/2)

- On peut écrire des programmes “plus précis” :

```
type term α =
| Int : forall α [ α = int ]. int → term α
| Succ : forall α [ α = int ].
    term α → term α
| Pair : forall α 'b 'c [ α = 'b × 'c].
    term 'b × term 'c → term α
| Fst : forall α 'b 'c [ α = 'b × 'c].
      term α → term 'b
| Snd : forall α 'b 'c [ α = 'b × 'c].
      term α → term 'c
```

```
let rec eval (t : term α) : α =
  match t with
  | Int i → (* α = int *) i
  | Succ t → (* α = int *) eval t + 1
  | Pair (t1, t2) →
      (* ∃βγ.α = β * γ *)
       (eval t1, eval t2)
  | Fst t →
      (* ∃βγ.α = β * γ *)
      fst (eval t)
  | Snd t →
      (* ∃βγ.α = β * γ *)
      snd (eval t)
```

- Grâce à ce typage plus précis, on a supprimé des *tests dynamiques*.

À retenir sur les types algébriques généralisés

On peut oublier *temporairement* la nature d'un type
grâce à un *témoin observable* dans le futur.

Applications

- Les types algébriques généralisés permettent de coder des contraintes de types entre des objets.
- Ces dernières augmentent la *sûreté* et l'*efficacité* des programmes.

Des analyseurs syntaxiques sûrs et efficaces

- Il existe des outils qui génèrent, à partir d'une grammaire LR, un programme qui simule l'exécution de l'automate à pile déterminististe correspondant.
- Comment garantir la *sûreté* du programme généré sans avoir à faire confiance à l'outil dont il est issu ?
- *François Pottier and Yann Régis-Gianas. Towards efficient, typed LR parsers*. In ACM Workshop on ML, volume 148(2) of Electronic Notes in Theoretical Computer Science, pages 155-180, March 2006.

Que font les outils existants ?

- Yacc, Bison, OCamlYacc etc. produisent des programmes C, sans *aucune garantie de sûreté.* Ils utilisent une *union* pour représenter les valeurs sémantiques et ne protègent pas contre les épuisements de pile.

- ML-Yacc et Happy produisent des programmes ML ou Haskell qui sont typés. Ils implémentent les valeurs sémantiques en utilisant une *union étiquetée*. Cependant, des *exceptions peuvent être lancées* lorsque une analyse de motifs échoue.

Une grammaire jouet

- Voici une grammaire LR très simple, tirée du *Dragon Book* :

$$
\begin{array}{lrcl}
(1) & E\{x\} + T\{y\} & \rightarrow & E\{x+y\} \\
(2) & T\{x\} & \rightarrow & E\{x\} \\
(3) & T\{x\} * F\{y\} & \rightarrow & T\{x \times y\} \\
(4) & F\{x\} & \rightarrow & T\{x\} \\
(5) & (E\{x\}) & \rightarrow & F\{x\} \\
(6) & \textbf{int}\{x\} & \rightarrow & F\{x\}
\end{array}
$$

- Les terminaux (ou *tokens*) sont +, *, (,), et **int**. Les *non-terminaux* sont E, T, et F. Les quatres premiers n'ont pas de valeurs sémantiques ; les quatres derniers sont associés à une valeur sémantique entière.

Interface de l'analyseur lexical

- Les *tokens* sont formés d'une étiquette et d'une éventuelle valeur sémantique :

```
type token = KPlus | KStar | KLeft | KRight | KEnd | KInt of int
```

- L'analyseur lexical fournit deux fonctions pour observer ou jeter le *token* courant :

```
val peek : unit → token
val discard : unit → unit
```

Structure de données pour l'automate

- Le type des états est aisément définissable :

```
type state = S0 | S1 | ... | S11
```

Structure de données pour la pile

- Le type pour la pile énumère les différents couples (symbole, état, valeur sémantique) empilables :

```
type stack =
    | SEmpty
    | SP of stack × state
    | SS of stack × state
    | SL of stack × state
    | SR of stack × state
    | SI of stack × state × int
    | SE of stack × state × int
    | ST of stack × state × int
    | SF of stack × state × int
```

- (P, S, L, R, I sont des raccourcis pour Plus, Star, Left, Right et Int.)

Implémentation (structure générale)

- L'automate est simulé par la fonction run. À partir de l'état courant, de la pile courante et du flux de tokens, cette fonction produit une valeur sémantique ou échoue.

```
let rec run (s : state) (stack : stack) : int =
  match s, peek() with
  | ... (* shift or reduce transitions *)
  | _, _ →
      raise SyntaxError
```

Implémentation (shift)

- Une transition *shift* pousse l'état courant et la valeur sémantique du token courant au sommet de la pile, jette le token courant, et on change l'état courant :

```
let rec run (s : state) (stack : stack) : int =
  match s, peek() with
  | ..
  | S9, KStar → (* shift S7 *)
      discard();
      run S7 (SS (stack, S9))
  | ...
```

Implémentation (reduce)

- Une transition *reduce* éjecte un nombre fixé de valeurs sémantiques de la pile et les utilise pour en calculer une nouvelle. qui est à son tour poussée au sommet de la pile.

```
let rec run (s : state) (stack : stack) : int =
  match s, peek() with
  | ...
  | S9, KPlus → (* reduce E{x} + T{y} → E{x + y} *)
      let ST (SP (SE (stack', s', x), _), _, y) = stack in
      let stack'' = SE (stack', s, x + y) in
      gotoE s' stack'' (* goto E *)
  | ...
```

- Observez que *l'analyse de motifs n'est pas exhaustive*.

Implémentation (fin)

- Une transition *goto* examine l'état qui a été ejecté de la pile durant la dernière réduction et change d'état courant.

```
and gotoE (s : state) : stack → int =
  match s with
  | S0 →
      run S1
  | S4 →
      run S8
```

- Encore une fois, *l'analyse de motifs est non exhaustive*.

En résumé

- Ce programme est considéré bien typé par un compilateur ML. Cependant, le compilateur produit des avertissements liés à la non exhaustivité des analyses de motifs, ce qui signifie que *l'absence d'erreurs durant l'exécution n'est pas garanti* !
- Le problème est de modifier ce programme de façon à ce que tous les analyses de motifs deviennent exhaustives. Supprimer ces tests dynamiques implique alors une *garantie de sûreté* et une plus *grande efficacité*.

Pourquoi ces tests sont-ils redondants ?

- Les tests dynamiques effectués durant la précédente transition de réduction sont redondants puisque, quand l'automate est dans l'état S_9, la pile doit être de la forme :

$$\ldots \quad S_? \quad E \quad S_? \quad + \quad S_? \quad T$$

- Les tests dynamiques effectués durant l'action *goto* E sont redondants puisque, lorsque l'automate est dans l'état S_9, la pile doit être de la forme :

$$\ldots \quad (S_0 \mid S_4) \quad ? \quad S_? \quad ? \quad S_? \quad ?$$

Un (fragment de l')invariant

- En fait, on peut prouver que, lorsque l'automate est dans l'état S_9, l'état doit être de la forme

$$\ldots \quad (S_0 \mid S_4) \quad E \quad (S_1 \mid S_8) \quad + \quad S_6 \quad T$$

  La forme de la pile peut ainsi être connue pour tout état.
- Cet *invariant* est prouvé par induction durant l'exécution de l'automate. Nous voulons que ce soit le vérificateur de type qui fasse cette preuve ...

L'idée

- On doit expliquer au compilateur qu'il existe une *corrélation* entre l'état courant de l'automate et la structure de sa pile.
- À cette fin, on rajoute un paramètre α au type state. L'idée est de coder la propriété :

Si l'état courant de l'automate est α state,
alors la pile courante de l'automate est de type α.

La structure de piles

- Le type stack *disparaît*. La structure des piles est définie par une famile de types paramétrés qui sont indépendantes les unes des autres :

$$
\begin{array}{l}
\textbf{type}\ \mathsf{empty} = \mathsf{SEmpty} \\
\textbf{type}\ \alpha\ \mathsf{cP} = \mathsf{SP}\ \textbf{of}\ \alpha \times \alpha\ \mathit{state} \\
\textbf{type}\ \alpha\ \mathsf{cS} = \mathsf{SS}\ \textbf{of}\ \alpha \times \alpha\ \mathit{state} \\
\textbf{type}\ \alpha\ \mathsf{cL} = \mathsf{SL}\ \textbf{of}\ \alpha \times \alpha\ \mathit{state} \\
\textbf{type}\ \alpha\ \mathsf{cR} = \mathsf{SR}\ \textbf{of}\ \alpha \times \alpha\ \mathit{state} \\
\textbf{type}\ \alpha\ \mathsf{cI} = \mathsf{SI}\ \textbf{of}\ \alpha \times \alpha\ \mathit{state} \times \mathsf{int} \\
\textbf{type}\ \alpha\ \mathsf{cE} = \mathsf{SE}\ \textbf{of}\ \alpha \times \alpha\ \mathit{state} \times \mathsf{int} \\
\textbf{type}\ \alpha\ \mathsf{cT} = \mathsf{ST}\ \textbf{of}\ \alpha \times \alpha\ \mathit{state} \times \mathsf{int} \\
\textbf{type}\ \alpha\ \mathsf{cF} = \mathsf{SF}\ \textbf{of}\ \alpha \times \alpha\ \mathit{state} \times \mathsf{int}
\end{array}
$$

Codé un (fragment de l') invariant

- Le fait que, quand l'automate est dans l'état S_9, la pile doit être de la forme

$$\ldots \quad S_? \quad E \quad S_? \quad + \quad S_? \quad T,$$

  est codé en affectant au constructeur de donnée S9 le type

$$\forall \alpha.\alpha\ cE\ cP\ cT\ state$$

  et ainsi de suite pour chaque état.
- Une telle déclaration est impossible en ML !
- Le type state est un *type algébrique généralisé* (GADT).

La structure des états

type state : $\star \longrightarrow \star$ where
| S0 : *empty state*
| S1 : *empty cE state*
| S2 : $\forall\alpha.\alpha$ *cT state*
| S3 : $\forall\alpha.\alpha$ *cF state*
| S4 : $\forall\alpha.\alpha$ *cL state*
| S5 : $\forall\alpha.\alpha$ *cI state*
| S6 : $\forall\alpha.\alpha$ *cE cP state*
| S7 : $\forall\alpha.\alpha$ *cT cS state*
| S8 : $\forall\alpha.\alpha$ *cL cE state*
| S9 : $\forall\alpha.\alpha$ *cE cP cT state*
| S10 : $\forall\alpha.\alpha$ *cT cS cF state*
| S11 : $\forall\alpha.\alpha$ *cL cE cR state*

Implémentation (structure générale)

- Seul le type de la fonction run change : il accepte maintenant un état arbitraire et une pile *dont la structure doit être cohérente vis-à-vis de cet état* :

```
let rec run : ∀α.α state → α → int =
  fun s stack →
    match s, peek() with
    | ...
    | _, _ →
        raise SyntaxError
```

(Comparer au type original.)

Implémentation (reduce)

- Le code des transitions *reduce* est inchangé, mais l'*analyse de motifs est maintenant exhaustive* :

```
let rec run : ∀α.α state → α → int =
  fun s stack →
    match s, peek() with
    | S9, KPlus →
        (* Il existe un type β inconnu tel que : *)
        (* S9 : β cE cP cT state *)
        (* s : β cE cP cT state *)
        (* α = β cE cP cT *)
        (* Thus stack : β cE cP cT *)
        let ST (SP (SE (stack', s', x), _), _, y) = stack in
        ...
```

Pour résumer

- Nous avons codé une partie de l'invariant de l'automate dans les déclarations de type et dans les types associés à run et goto.
- En fait, la totalité de l'invariant peut ainsi être codé dans les types (voir le papier).

Aller plus loin . . .

- Les GADT offre au programmeur la possibilité de *raisonner* sur les types.
- Néanmoins, ce raisonnement est essentiellement *syntaxique*.
- En effet, on manipule uniquement des *égalités entre types*.

$\Rightarrow$ Comment augmenter l'expressivité de ce système ?

$\Rightarrow$ Peut-on effectuer des raisonnements *arbitraires* sur les types ?

Un détour par les années 60

- Méthode de Hoare :
  1. Annotez vos programmes par des assertions logiques.
  2. Calculez les théorèmes à prouver.
  3. Prouvez !

Exemple : recherche d'un entier dans un tableau d'entiers

```
/* Proved using Caduceus (http ://why.lri.fr) */
/*@ requires \valid_range(t,0,n-1)
  @ ensures
  @ (0 ≤ \result ⟨ n =⟩ t[\result] == v) ∧
  @ (\result == n =⟩ \forall int i ; 0 ≤ i ⟨ n =⟩ t[i] != v)
  @*/
int index(int t[], int n, int v) {
  int i = 0 ;
  /*@ invariant 0 ≤ i ∧ \forall int k ; 0 ≤ k ⟨ i =⟩ t[k] != v
    @ variant n - i */
  while (i ⟨ n) {
    if (t[i] == v) break ;
    i++ ;
  }
  return i ;
}
```

Exemple : recherche d'un entier dans un tableau d'entiers

```
/* Proved using Caduceus (http ://why.lri.fr) */
/*@ requires \valid_range(t,0,n-1)
  @ ensures
  @ (0 ≤ \result ⟨ n =⟩ t[\result] == v) ∧
  @ (\result == n =⟩ \forall int i ; 0 ≤ i ⟨ n =⟩ t[i] != v)
  @*/
int index(int t[], int n, int v) {
  int i = 0 ;
  /*@ invariant 0 ≤ i ∧ \forall int k ; 0 ≤ k ⟨ i =⟩ t[k] != v
    @ variant n - i */
  while (i ⟨ n) {
    if (t[i] == v) break ;
    i++ ;
  }
  return i ;
}
```

Des égalités de type dans les formules logiques ?

- À l'aide d'information sur les types issue d'un raisonnement arbitraire, on pourrait imaginer traiter les dépassements de capacité statiquement :

  let add (x : α) (y : 'b) : 'c =
  where $\{$
  max_capacity α + max_capacity 'b $\leq$ max_capacity 'c **and**
  numerics α **and** numerics 'b **and** numerics 'c
  $\}$ =
  ...

- avec

  (Max capacity α is the biggest number storable into α *)*
  max_capacity : **type** $\rightarrow \mathbb{N}$

  (A predicate over types that characterizes data representation that can be used in numerical operations. *)*
  numerics : $\alpha \rightarrow$ prop

À suivre . . .

Conclusion

- Le typage fournit des *garanties statiques* automatiquement.
- Il augmente la *généralité* et l'*efficacité* des programmes.
- La close interaction entre types et formules logiques favorise le *passage à l'échelle de preuve de propriétés arbitrairement complexes* sur les programmes.

Merci. Des questions ?