*Retour sur le raisonnement sur les programmes*

## L'égalité sur les programmes

- *une notion d'égalité sur les programmes?*
  - on sait raisonner en maths sur $aX^2+bX+c=0$, et sur des équations bien plus compliquées
  - que signifie écrire (voire résoudre) des équations entre programmes?
  - on veut définir une relation $=$ entre programmes (ou expressions), $= \subseteq \mathrm{Caml}^2$
- on souhaite les propriétés usuelles
  - $=$ est réflexive symétrique transitive
  - $=$ est une *congruence*
    - l'égalité peut être utilisée "partout", pour remplacer une chose par une chose égale (comme toujours en maths)
    - si `e1`=`e2`, alors tout programme `P` est égal à `P` dans lequel on a remplacé `e1` par `e2`

## Egalité sur les programmes, suite

- $=$ est définie *"hors de Caml"*
  - `(=) : 'a -> 'a -> bool` ne dit pas assez
    ne permet pas de comparer des fonctions
  - `(==) : 'a -> 'a -> bool` est trop discriminant
    on veut `[1;2]` $=$ `[1;2]` et `quicksort` $=$ `insertsort`
  - `=` et `==` ne sont définies qu'entre *valeurs*
    on aimerait dire `(fact 3)` $=$ `6` $=$ `7-1`
- deux manières de parler de l'exécution des programmes
  - à petits pas: `P` $\rightarrow$ `P'`  `P` et `P'` expressions
  - à grands pas: eval(`P`) $=$ `V`  `V` valeur
- quelles égalités?
  - on veut $\rightarrow \subseteq =$, i.e., si `P` $\rightarrow$ `P'`, alors `P` $=$ `P'`
    ça ne suffit pas: `3+2` $=$ `6-1`, et pourtant `3+2` $\not\rightarrow$ `6-1`

## Equations entre programmes

- posons `P`=`P'` si eval(`P`)=eval(`P'`)
  ou plutôt si eval(`P`)=eval(`P'`)
- comment définir `f` $=$ `g` pour `f` et `g` deux fonctions?
  égalité *extensionnelle*:
  `f`$=_{\mathrm{E}}$`g` ssi pour tout `x` (du bon type), `f x` $=$ `g x`
  ou plutôt eval(`f x`)=eval(`g x`)
- la condition de congruence: égalité *contextuelle*
  `f`=`g` ssi pour tout *contexte* `e[]`, eval(`e[f]`)=eval(`e[g]`)
  - un contexte est un programme (une expression) avec un emplacement vide
  - `e[f]` désigne l'expression obtenue en mettant `f` à la place de l'emplacement vide
    `...f...f...` $\rightsquigarrow$ `...g...g...`
  - $=_{\mathrm{E}}$, c'est $=$ avec les contextes `([] x)` pour tout `x`
- est-ce que $=_{\mathrm{E}}$ implique $=$?

## Egalité entre programmes

- on a défini une *égalité sur les programmes*, $=$
  - on impose que $=$ soit une **congruence**
  - `e`=`e'` ssi eval(`e`)=eval(`e'`)
    - ok tant que le type de `e` n'est pas un type flèche (`t1->t2`)
    - cela fait sens car l'*évaluation est déterministe*
- la grande question: les fonctions
  - égalité extensionnelle:
    `f`$=_{\mathrm{E}}$`g` ssi pour tout `x` du bon type, `(f x)`=`(g x)`
  - a-t-on (`f`$=_{\mathrm{E}}$`g`) $\Rightarrow$ (`f`=`g`) ?
    - `let f1 = fun x -> x+1` et `let f2 = fun x -> (y:=0; x+1)`
    - `f1`$=_{\mathrm{E}}$`f2`, mais `f1`$\neq$`f2`, avec `let t f = (y:=1; y:= (f !y); !y)`
    - oui mais la seconde fonction mentionne `y` et pas la première

## Egalités entre fonctions, avec des `ref`

- autre exemple

```
let f1 =                          let f2 =
  let a = ref () in                 let c = ref () in
  let b = ref () in                 let d = ref () in
fun x -> if x==a then b else a    fun y -> if y==d then d else c
```

  - `f1`$\neq$`f2`

```
let t = fun f -> let x = ref () in f (f x) == f x
```

  - au passage, `ref ()` est la "parcelle d'identité élémentaire" en Caml
- on peut montrer `f1`=`f2` avec

```
let f1 =                          let f2 =
  let a = ref 0 in                  let b = ref 0 in
  fun x -> (a := !a+x; !a)          fun y -> (b := !b-y; 0-!b)
```

- c'est un sujet de recherche
  - comment définir une égalité "robuste" entre programmes?
  - comment prouver que deux programmes sont équivalents?

## Effets de bord et raisonnement formel

- `f1` $=_{\mathrm{E}}$ `f2` $\Rightarrow$ `f1` $=$ `f2`  est vrai en absence d'effets de bord
  - on ne verra pas cela en cours
- plus généralement, les effets de bord rendent beaucoup d'égalités caduques
  - ainsi, `let r = ref 1 in r := 2; !r` $\neq$ `(ref 1) := 2; !(ref 1)`
  - alors que l'on souhaiterait avoir l'égalité

    `let x = e1 in e2` $=$ `e2{e1/x}`

    - `e2{e1/x}` : `e2` où `x` est remplacé par `e1`
    - attention: il faut parfois renommer ($\alpha$-conversion)
      `let x = (g 32) in (fun x -> x+3) (x*x)`
      $\neq$ `(fun x -> (g 32)+3) ((g 32)*(g 32))`

## Transparence et pureté

- une expression est *"référentiellement transparente"* si on peut la remplacer par sa valeur sans changer le sens du programme global
  - i.e., eval(`P`)=`P`
  - `fun x -> y:=0; 0`  ok
  - `(fun x -> y:=0; 0) 3`  non
- lié à la notion de fonction *pure*: pas d'effet de bord lors de l'évaluation
  - permet d'optimiser lors de la compilation
    p.ex. `f(x) * f(x)` remplacé par `let y = f(x) in y*y`
  - compilateur gcc: attribut `pure` pour signaler les fonctions pures
    - le compilateur détecte tout seul les fonctions pures
    - mais avec des librairies précompilées, on n'a pas cette information