Cours de Programmation Avancée

L3 ENS Paris Saclay

Giuseppe Castagna

CNRS

Outline of the course

Le langage de référence pour le cours est OCaml, mais nous utiliserons aussi des extraits de code Haskell, Scala, Perl 6, C#, Java, Erlang, Pascal, Python, Basic, CDuce, Xslt, Go ... . L'idée étant de mettre l'accent sur les concepts de la programmation plus que sur la programmation dans un langage particulier.

La note finale de l'examen est calculée de la manière suivante.

(1/2 (Examen) + 1/3 max(Examen, Projet) + 1/6 Projet)*CB

[CB = if Projet > 6 then 1 else 0.7]

Modules

Outline

Outline

Les modules sont partout !

- Toute construction complexe :
  - bâtiment,
  - voiture,
  - avion,
  - compilateur,. . .

  suppose une *modularisation*.
- Les raisons technologiques sont évidentes (organisation du travail, fiabilité,. . .)
- . . . et les retombées économiques immédiates.

Généralités sur les modules

- On peut construire, tester, analyser. . . un module de façon *indépendante* d'autres modules.
- Un module a une *interface* qui décrit ses modalités d'interaction.
- Éventuellement, une *spécification* qui décrit son comportement.
- Une *implémentation*.

Un changement d'implémentation devrait être *"transparent"* à l'utilisateur (à un certain niveau d'abstraction !).

Aspects spécifiques des modules en programmation

Découpage en *unités logiques* plus petites

But :

réalisation d'un module séparément des autres modules

Mise en œuvre :

un module possède une interface, la vérification des interfaces est effectuée à l'assemblage des différents modules.

Intérêts :

- découpage logique ;
- abstraction des données (spécification et réalisation) ;
- indépendance de l'implémentation ;
- réutilisation.

Compilation séparée

Découpage en *unités de compilation*, compilables séparément

programmation modulaire $\neq$ compilation séparée

les 2 approches sont nécessaires :

- Pour cela la spécification d'un module doit être vérifiable par un compilateur :
  - on se limite à la vérification de types
  - l'interface sera spécification de modules
  - et contiendra l'information de typage et de compilation pour les autres modules

Programmation à petite et grande échelle

- A petite échelle (Wirth 1975) :

  Programme = Algorithme + Structures de Données

- A grande échelle :

  Module = Programme + Interface + (Spécification)

Un problème ancien…

Le problème dans le contexte de la programmation a été identifié depuis longtemps. Par exemple :

> *DeRemer, Kron. Programming in the large versus programming in the small. IEEE Trans. on Soft. Eng., 1976.*
>
> *Parnas. On the criteria to be used in decomposing systems into modules. CACM, 1972.*

Un problème toujours d'actualité. . .

- Bibliothèques,
- "Modules" en C.
- Packages en Ada et Java.
- Modules en Modula et ML.
- Programmation par composants
- Interfaces 'Web services'.

Un problème plus difficile : l’interopérabilité

Programmation modulaire On cherche à faire interagir des modules qui appartiennent au *même langage*.

Interopérabilité On cherche à faire interagir des modules de langages qui *diffèrent* dans :

- la représentation des données,
- le traitement des exceptions,
- l’organisation de la mémoire,
- ...

Outline

Les modules en ML

Nous allons étudier le système de modules de ML.

- De loin le système de modules le plus sophistiqué et le plus étudié.
- La conception du système de modules est assez *indépendant* du langage de programmation. Le système de modules de ML a été appliqué aussi à d'autres langages.
- Une généralisation du concept de type de données abstrait.

Théorie (des types) encore en développement : compliquée et pas stable. Les détails présentés ici sont basés sur OCAML et en particulier sur le Chapitre 14 de
`https://caml.inria.fr/pub/docs/oreilly-book/html/index.html`

Terminologie ML

- Structure (= Implémentation).
- SIGNATURE (= Interface).

$$\textit{Structure} : \textit{SIGNATURE} \quad \sim \quad \textit{Valeur} : \textit{TYPE}$$

Remarques

1. En ML on sépare *valeurs* et *types*. Or les structures contiennent des *types* et des *valeurs*, on ne peut donc pas les voir comme des valeurs.
2. Une *structure* (une *signature*) n'est pas une valeur (un type) de première classe.

Modules (simples)

Structure

Une suite de définitions de :

- valeurs (y compris fonctions)
- types
- exceptions
- sous-modules

SIGNATURE

Une suite déclarations de types, d'exceptions et de noms avec leur type/signature.

Convention

On utilise, `Machin` pour une structure et `MACHIN` pour une signature.

Exemple : Structure d’un module Queue

```
module Queue =
struct
  type 'a queue = 'a list ref
  let create() = ref []
  let enq x q = q:= !q@[x]        (* horrible si @ n'est pas lazy *)
  let deq q =
    match !q with
         [] -> failwith "Empty"
       | h::r -> q:=r; h
  let length q = List.length !q       (* utilisation module List *)
end ;;
```

La signature associée

La système synthétise automatiquement la signature suivante.

```
module Queue :
sig
  type 'a queue = 'a list ref
  val create : unit -> 'a list ref
  val enq : 'a -> 'a list ref -> unit
  val deq : 'a list ref -> 'a
  val length : 'a list ref -> int
end
```

Dans la signature générée le type de données qui représente la queue est *visible* ainsi que l'ensemble des opérations définies avec les type le plus général.

Un autre exemple avec structures imbriquées

Valeur (structure)

```
module Example =
struct
  type t = int
  module M =
  struct
    let succ x = x+1
  end
  let two = M. succ(1);;
end;;
```

Type (signature)

```
module Example :
sig
  type t = int
  module M :
  sig
    val succ : int -> int
  end
  val two : int
end;;
```

Accès aux éléments d’un module par identificateur qualifié

- La notation ‘point’ :

```
# Queue.enq;;
- : 'a -> 'a list ref -> unit = <fun>
```

- S’applique aussi aux champs d’enregistrements :

```
# module Toto = struct type t = {x:int; y:int} end;;
module Toto : sig type t = { x: int; y: int } end

# let u = {Toto.x=3; Toto.y=18};;
val u : Toto.t = {Toto.x=3; Toto.y=18}
```

Ouverture d’un module

- On peut ouvrir un module et donc accéder toutes les déclarations de la structure associée :

```
# open Queue;;
# let q = create() in ( enq "Bob" q; q);;
- : string list ref = {contents = ["Bob"]}

# Example.two;;
-: int = 2

# Example.M.succ;;
-: int -> int = <fun>

# Example.M.succ (Example.two);;
-: int = 3

(* une structure n’est pas une valeur *)
# Example.M;;
Error: Unbound constructor Example.M
```

L’ouverture d’un module peut cacher des déclarations locales.

Déclaration d’une signature

- On peut déclarer une *signature* comme suit :

```
module type QUEUE =
 sig
 type 'a queue = 'a list ref
 val create : unit -> 'a list ref
 val enq : 'a -> 'a list ref -> unit
 val deq : 'a list ref -> 'a
 val length : 'a list ref -> int
end;;
```

- En associant une structure à une signature le système vérifie que tous les éléments de la signature existent dans la structure. Par exemple :

```
 module Queue : QUEUE = struct...end;;
```

- La structure peut contenir un élément avec un type *plus général* que celui spécifié dans la signature.
- Elle peut aussi contenir *d'avantage d'éléments* que ceux décrits dans la signature et dans un ordre différent.

```
module Example = struct
  type t = int
  module M = struct
   let succ x = x+1
  end
  let two = M.succ(1)
end ;;

module type ABS = sig
  type t
  val two : t
end;;
```

- Il s'agit d'un exemple de *sous-typage* (sur les signatures). On discutera ce concept plus tard dans le cours.

- En associant une structure à une signature on ne peut utiliser que les éléments déclarés dans la signature. Ici un type *t* dont on ignore la représentation et une valeur *two* de ce type.
- Le module Abs est une restriction du module Example.

```
# module Abs = (Example: ABS);; (* Nous cachon M et t *)
module Abs : ABS

# Abs.two;;                     (* M est utilisable *)
- Abs.t = <abstr>               (* t est abstrait   *)

# Abs.M.succ(1)                 (* M est invisible  *)
Unbound value Abs.M.succ
```

Types concrets et types abstraits

On définit une signature LISTE.

```
# module type LISTE = sig
    type 'a t
    val creer : unit -> 'a t
    val inserer : 'a -> 'a t -> 'a t
  end;;
```

dont on propose deux *implémentations* :

```
# module Liste1 = struct
    type 'a t = 'a list
    let creer () = []
    let inserer x l = x::l
  end;;

module Liste1 : sig
  type 'a t = 'a list
  val creer : unit -> 'a list
  val inserer : 'a -> 'a list -> 'a list
end
```

```
# module Liste2 = struct
    type 'a t = 'a list
    let creer () = []
    let inserer x l = l (*bad*)
  end;;

module Liste2 : sig
  type 'a t = 'a list
  val creer : unit -> 'a list
  val inserer : 'a -> 'b -> 'b
end
```

La structure concrète des types est *visible* et on peut écrire :

```
# Liste1.creer();;
- : 'a list = []

# Liste2.inserer 3 (Liste1.creer ()) ;;
- : 'a list = []
```

Maintenant on *masque* Liste1 et Liste2 avec la signature LISTE :

```
# module Liste1 = (Liste1: LISTE);;
module Liste1 : LISTE

# module Liste2 = (Liste2: LISTE);;
module Liste2 : LISTE
```

Les types deviennent *abstraits* et donc *incomparables*.

```
# Liste1.creer();;
- : '_a Liste1.t = <abstr>

#  Liste2.inserer;;
- : 'a -> 'a Liste2.t -> 'a Liste2.t = <fun>

#  Liste2.inserer 3 (Liste1.creer ()) ;;
This expression has type 'a Liste1.t
but is here used with type int Liste2.t
```

Variables de types faibles

Une variable de type faible `'_a` est une variable qui ne peut pas être généralisée. Elle est instanciée par un type qui n'a pas encore pu être déterminé.

Elle apparait lorsque le compilateur Caml essaie de compiler une fonction ou une valeur qui est monomorphe, mais pour laquelle certains types n'ont pu être complètement inferés.
Elles disparait grâce au mécanisme d'inférence de types dès que suffisamment d'informations auront pu être rassemblées.

```
# let id x = x;;
val id : 'a -> 'a = <fun>

# let id2 = id id;;
val id2 : '_a -> '_a = <fun>

# let a = (id 1 , id "1");;
val a : int * string = (1, "1")

# let b = (id2 1 , id2 "1");;
Error: This expression has type string but an
       expression was expected of type int
```

Le problème vient de l'utilisation des références mutables :

```
# let r = ref []
val r : 'a list ref
# r := [3]; r
- : 'a list ref
# let l = List.map (function true -> 1 | false -> 2) !r
val l : int list = Segmentation fault
```

Solution in ML (Wright '95) : seulement les *valeurs* peuvent être polymorphes (i.e., pas les applications ni les créations de cellules de mémoire).

```
# let r = ref [] ;;
val r : '_a list ref
# r := [3]; r ;;
- : int list ref
# let l = List.map (function true -> 1 | false -> 2) !r;;
Error: This expression has type int list
       but an expression was expected of type bool list
```

Révenons à l’example Queue

Une manière plus raisonnable de *déclarer la signature* de Queue

```
module type QUEUE = sig
 type 'a queue
 exception Empty
 val create : unit -> 'a queue
 val enq : 'a -> 'a queue -> unit
 val deq : 'a queue -> 'a
 val length : 'a queue -> int
end;;
```

```
module Queue : QUEUE = struct
  type 'a queue = 'a list ref
  exception Empty
  let create() = ref []
  let enq x q = q:= !q@[x]
  let deq q =
    match !q with
          [] -> raise Empty
        | h::r -> q:=r; h
  let length q = List.length !q
end ;;
```

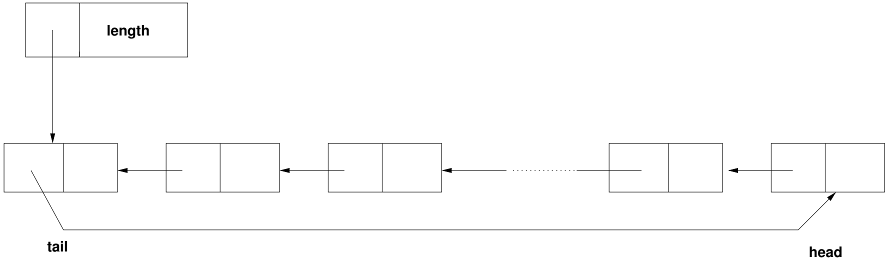
(nous avons caché que le type utilisé pour l’implémentation est `'a list ref`)

Une manière plus raisonnable de *définir* Queue

```
module Queue : QUEUE = struct
  type 'a cell = { content: 'a;  mutable next: 'a cell }    (* invisible *)
  type 'a queue = { mutable length: int; mutable tail: 'a cell }
  exception Empty
  let create () = { length = 0; tail = Obj.magic None }
  let enq x q =
    q.length <- q.length + 1;
    if q.length = 1 then
       let rec cell = { content = x; next = cell} in
       q.tail <- cell
    else
       let tail = q.tail in
       let head = tail.next in
       let cell = { content = x; next = head} in
       tail.next <- cell;
       q.tail <- cell
  let deq q =
    if q.length = 0 then raise Empty;
    q.length <- q.length - 1;
    let tail = q.tail in
    let head = tail.next in
    if head == tail then
      q.tail <- Obj.magic None
    else
      tail.next <- head.next;
    head.content
  let length q = q.length
end;;
```

Voyons cela graphiquement

Implementation de Queue



L'utilisation de option est plus propre mais 5% plus lente

```
module Queue : QUEUE = struct
  type 'a cell = { content: 'a;  mutable next: 'a cell option}
  type 'a queue = { mutable length: int; mutable tail: 'a cell }
  exception Empty
  let create () = { length = 0; tail = None }
  let enq x q =
    q.length <- q.length + 1;
    if q.length = 1 then
       let rec cell = { content = x; next = cell} in
       q.tail <- Some cell
    else
       let Some tail = q.tail in      (* non exhaustive pattern matching *)
       let head = tail.next in
       let cell = { content = x; next = head} in
       tail.next <- cell;
       q.tail <- Some cell
  let deq q =
    if q.length = 0 then raise Empty;
    q.length <- q.length - 1;
    let Some tail = q.tail in         (* non exhaustive pattern matching *)
    let head = tail.next in
    if head == tail then
      q.tail <- None
    else
      tail.next <- head.next;
    head.content
  let length q = q.length
end;;
```

Partage de type par contrainte

L'abstraction peut être une source de difficultés...

- Par masquage on peut rendre un type *abstrait*.
- A priori tous les types abstraits sont *différents*.
- Parfois, on souhaite spécifier que deux types abstraits sont le même de façon à les *partager* entre plusieurs modules.
- Le langage offre la possibilité d'exprimer des contraintes d'*égalité de types*.

Exemple (partage de types)

- Un module M avec un type abstrait t.

```
# module M =
 (
   struct
     type t = int ref
     let create() = ref 0
     let add x = incr x
     let get x = if !x>0 then (decr x; 1) else failwith "Empty"
   end
   :
   sig
     type t
     val create : unit -> t
     val add : t -> unit
     val get : t -> int
   end
 ) ;;
```

- On restreint la vue du module `M` de deux façons.

```
# module type S1 =
   sig
     type t
     val create : unit -> t
     val add : t -> unit
   end ;;

# module type S2 =
   sig
     type t
     val get : t -> int
   end ;;

# module M1 = (M:S1) ;;
module M1 : S1

# module M2 = (M:S2) ;;
module M2 : S2
```

- Le problème est que les types `M1.t` et `M2.t` ne sont pas identifiés :

```
# let x= M1.create ();;
val x : M1.t = <abstr>

# M1.add x;;
- : unit = ()

# M2.get x;;
This expression has type M1.t but is here used with type M2.t
```

- On règle le problème avec des contraintes d'égalité

```
# module M1 = (M:S1 with type t = M.t) ;;
module M1 : sig
   type t = M.t
   val create : unit -> t
   val add : t -> unit
end

# module M2 = (M:S2 with type t = M.t) ;;
module M2 : sig
   type t = M.t
   val get : t -> int
end

# let x = M1.create() in M1.add x ;  M2.get x ;;
- : int = 1
```

Une solution alternative avec sous-modules

- On construit M1 et M2 comme *sous-modules* de M

```
# module M =
  (struct
     type t = int ref
     module M_hide =
       struct
         let create() = ref 0
         let add x = incr x
         let get x = if !x>0 then (decr x; 1) else failwith"Empty"
       end
     module M1 = M_hide
     module M2 = M_hide
   end
  :
   sig
     type t
     module M1 : sig  val create : unit -> t  val add : t -> unit end
     module M2 : sig  val get : t -> int  end
   end ) ;;
```

- Le type synthétisé est :

```
module M :
  sig
    type t
    module M1 : sig val create : unit -> t val add : t -> unit end
    module M2 : sig val get : t -> int end
  end
```

- Maintenant `M1` et `M2` font référence au même type abstrait.

```
# let x = M.M1.create() ;;
val x : M.t = <abstr>

# M.M1.add x ; M.M2.get x ;;
- : int = 1
```

- Ce n'est pas "modulaire" (conception de `M1`, `M2` a posteriori impossible)

- Avec les foncteurs (prochain argument) cette solution n'est plus viable

Inclusion de structures et signatures

Nous avons vu comment *restreindre* la vue d'une signature. Quid si on veut *élargir* une structure ou une signature ?

```
# module type S =
sig
  type t
  val x: t
  val f: t->t
end;;

# module type S1 =
sig
  include S          (* inclusion d'une signature *)
  val g: t->t
end;;

module type S1 = sig type t val x : t val f : t -> t val g : t -> t end
```

Un autre exemple avec Points et Cercles

```
module type POINT =
sig
  type point = float * float
  val mk_point: float * float -> point
  val x_coord:  point -> float
  val y_coord:  point -> float
  val move_p : point * float * float -> point
end;;

module type CIRCLE =
sig
  include POINT                               (* inclusion de la signature *)
  type circle
  val mk_circle: point * float -> circle
  val center:  circle -> point
  val radius:  circle -> float
  val move_c : circle * float * float -> circle
end;;
```

```
module Point: POINT =
struct
  type point = float * float
  let mk_point(x,y)  = (x,y)
  let x_coord(x,y) = x
  let y_coord(x,y) = y
  let move_p ((x,y),dx,dy) = (x+.dx,y+.dy)
end;;

module Circle: CIRCLE = struct
  include Point                     (* inclusion de la structure *)
  type circle = point * float
  let mk_circle (x,y) = (x,y)
  let center(x,y) = x
  let radius (x,y) = y
  let move_c(((x,y),r),dx,dy) = ((x+.dx,y+.dy),r)
end;;
```

Compilation séparée (1)

- Dans la programmation à grande échelle il convient de séparer un programme en plusieurs fichiers qui peuvent être compilés séparément.
- En OCAML, *unité de compilation = deux fichiers* :
  - le fichier implémentation `nom.ml` (= contenu d'une structure)
  - le fichier interface `nom.mli` (= contenu d'une signature)
  - Les deux fichiers sont équivalents à la déclaration

```
module Nom = (
   struct
     (* contenu de nom.ml *)
   end :
   sig
     (* contenu de nom.mli *)
   end
)
```

Compilation separeée (2)

Correspondance nom de module et nom de fichier :

- module `Nom` correspond aux fichiers `nom.ml` et `nom.mli`
- environnement de typage : répertoires d'accès aux fichiers
- Le fichiers `nom.ml` et `nom.mli` peuvent être compilés séparément avec l'option -c (compiler sans lier)

`% ocamlc -c aux.mli`	produit	`aux.cmi`	code objet interface
`% ocamlc -c aux.ml`	produit	`aux.cmo`	code objet implantation
`% ocamlc -c main.mli`	produit	`main.cmi`	code objet interface
`% ocamlc -c main.ml`	produit	`main.cmo`	code objet implantation
`% ocamlc aux.cmo main.cmo -o main`	linking		

- Le programme est équivalent à :

```
module Aux: sig (* contenu de aux.mli *) end
          = struct (* contenu de aux.ml *) end;;
module Main: sig (* contenu de main.mli *) end
           = struct (* contenu de main.ml *) end;;
```

En particulier `Main` peut faire référence aux définitions dans l'interface de `Aux`, mais `Aux` ne peut pas faire référence à `Main`.

- Depuis la version 3.07 de OCaml il est possible de définir des structures et des signatures récursives par la syntaxe :

  ```
  module rec ... and ...
  ```

  avec des restrictions pour assurer la terminaison :
  - Tout cycle de dépendance doit passer par au moins un module “safe".
  - Un module est “safe” si tout valeur défini dans le module est une fonction
  - L’évaluation démarre en construisant les modules “safe" dont les valeurs sont initialisées à `fun _ -> raise Undefined_recursive_module`.

```
module rec A : sig
                 type t = Leaf of string | Node of ASet.t
                 val compare: t -> t -> int
               end
             = struct
                 type t = Leaf of string | Node of ASet.t
                 let compare t1 t2 =
                   match (t1, t2) with
                     (Leaf s1, Leaf s2) -> Pervasives.compare s1 s2
                   | (Leaf _, Node _) -> 1
                   | (Node _, Leaf _) -> -1
                   | (Node n1, Node n2) -> ASet.compare n1 n2
               end
    and ASet : Set.S with type elt = A.t
             = Set.Make(A)
```

On peut lui donner la spécification suivante

```
module rec A : sig
                 type t = Leaf of string | Node of ASet.t
                 val compare: t -> t -> int
               end
               and ASet : Set.S with type elt = A.t
```

Outline

Fonteurs

- Les *foncteurs* sont des fonctions des structures dans des structures.
- Ils sont utilisés pour exprimer une structure qui dépend d'une autre structure.
- Comme pour les fonctions, on écrit une fois un code qui pourra être utilisé plusieurs fois.

Exemple

- On définit la signature d'un *groupe*.

```
# module type GROUPE =
sig
  type g
  val e: g
  val comp: g*g -> g
  val inv: g -> g
end;;
```

- On définit la construction de carré d'un groupe comme un 'foncteur' des groupes dans les groupes.

```
# module Square (Gr: GROUPE) =
  ( struct
     type g = Gr.g * Gr.g
     let e = (Gr.e,Gr.e)
     let comp ((a,b),(c,d)) = (Gr.comp(a,c),Gr.comp(b,d))
     let inv (a,b) = (Gr.inv(a),Gr.inv(b))
    end : GROUPE );;
module Square : functor (Gr : GROUPE) -> GROUPE
```

- On peut construire la structure GROUPE des entiers.

```
# module Zeta: GROUPE =
  struct
    type g = int
    let e = 0
    let comp (n,m) = n+m
    let inv (n) = -n
  end;;
```

- et générer le groupe carré par application.

```
# module SquareZeta = Square(Zeta) ;;
module SquareZeta :
  sig
    type g = Square(Zeta).g
    val e : g
    val comp : g * g -> g
    val inv : g -> g
  end
```

NB Ici le type dans le résultat est *abstrait*.

Un autre exemple : mots ordonnés

- On déclare une signature *type ordonné*.

```
type comparison = Less | Equal | Greater;;

module type ORDERED_TYPE =
    sig
     type t
     val compare: t -> t -> comparison
    end;;
```

- On déclare un foncteur Set qui est paramétré sur un type ordonné.

```
module Set (Elt: ORDERED_TYPE) =
  struct
    type element = Elt.t
    type set = element list
    let empty = []
    let rec add x s =
      match s with
        [] -> [x]
      | hd::tl ->
         match Elt.compare x hd with
           Equal   -> s          (* x is already in s *)
          | Less    -> x :: s     (* x is smaller than all elmts of s *)
          | Greater -> hd :: add x tl
    let rec member x s =
      match s with
        [] -> false
      | hd::tl ->
          match Elt.compare x hd with
            Equal   -> true      (* x belongs to s *)
          | Less    -> false     (* x is smaller than all elmts of s *)
          | Greater -> member x tl
  end;;
```

Le type inferé est :

```
module Set :
functor (Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      type set = element list
      val empty : 'a list
      val add : Elt.t -> Elt.t list -> Elt.t list
      val member : Elt.t -> Elt.t list -> bool
    end
```

- On construit la structure *mots ordonnés*.

```
# module OrderedString =
    struct
      type t = string
      let compare x y =
            if x = y then Equal
            else if x < y then Less
            else Greater
    end;;
module OrderedString :
  sig type t = string val compare : 'a -> 'a -> comparison end
```

- On dérive par *application* la structure *ensembles de mots*

```
# module StringSet = Set(OrderedString);;
module StringSet :
  sig
    type element = OrderedString.t
    type set = element list
    val empty : 'a list
    val add : OrderedString.t -> OrderedString.t list -> OrderedString.
    val member : OrderedString.t -> OrderedString.t list -> bool
  end

# StringSet.member "bar" (StringSet.add "foo" StringSet.empty);;
- : bool = false
```

Foncteurs et abstraction

- On souhaite cacher le fait que les ensembles sont représentés par des listes.
- On déclare une signature *de foncteur*.

```
# module type SETFUNCTOR =
  functor (Elt: ORDERED_TYPE) ->
    sig
      type element = Elt.t      (* concrete *)
      type set                  (* abstract *)
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end;;
```

- On utilise la signature pour créer une vue abstraite de Set.

```
# module AbstractSet = (Set : SETFUNCTOR);;
module AbstractSet : SETFUNCTOR

# module AbstractStringSet = AbstractSet(OrderedString);;
module AbstractStringSet :
   sig
     type element = OrderedString.t
     type set = AbstractSet(OrderedString).set    ←list n'est plus visible!
     val empty : set
     val add : element -> set -> set
     val member : element -> set -> bool
   end

#AbstractStringSet.add "gee" AbstractStringSet.empty;;
 - : AbstractStringSet.set = <abstr>
```

Foncteurs et abstraction (suite)

On considère un ordre non-standard sur le mots (on ne distingue pas lettres majuscules et minuscules).

```
# module NoCaseString =
  struct
    type t = string
    let compare s1 s2 =
      OrderedString.compare (String.lowercase s1) (String.lowercase s2)
  end;;
module NoCaseString :
    sig type t = string val compare : string -> string -> comparison end
```

On utilise le foncteur `AbstractSet` pour construire des ensembles de mots dont le type de representation est abstrait

```
 # module NoCaseStringSet = AbstractSet(NoCaseString);;
 module NoCaseStringSet :
    sig
      type element = NoCaseString.t
      type set = AbstractSet(NoCaseString).set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end
```

```
# NoCaseStringSet.add "FOO" AbstractStringSet.empty;;
  This expression has type
    AbstractStringSet.set = AbstractSet(OrderedString).set
  but is here used with type
    NoCaseStringSet.set = AbstractSet(NoCaseString).set
```

Les types AbstractStringSet.set et NoCaseStringSet.set sont *incompatibles*

Nota Bene

Ceci est *souhaitable*. Par exemple, l'union sur AbstractStringSet est différente de l'union sur NoCaseStringSet.set.

Foncteurs et Contraintes

On nomme `SET` la signature de la structure rendue par le foncteur AbstractSet.

```
# module type SET = sig
    type element
    type set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end;;
```

On pourrait penser d'utiliser `SET` pour abstraire le foncteur `Set`

```
# module WrongSet = (Set : functor(Elt: ORDERED_TYPE) -> SET);;
module WrongSet : functor (Elt : ORDERED_TYPE) -> SET

# module WrongStringSet = WrongSet(OrderedString);;
module WrongStringSet : sig
   type element = WrongSet(OrderedString).element
   type set = WrongSet(OrderedString).set
   val empty : set
   val add : element -> set -> set
   val member : element -> set -> bool
end

# WrongStringSet.add "gee" WrongStringSet.empty;;
This expression has type string but is here used with type
  WrongStringSet.element = WrongSet(OrderedString).element
```

Le problème est que SET spécifie le type des éléments de façon abstraite.
Ainsi WrongStringSet.element n'est pas le même type que string.
Pour surmonter cette difficulté on doit ajouter des *contraintes* :

```
# module AbstractSet =
   (Set : functor(Elt: ORDERED_TYPE) -> (SET with type element = Elt.t));;
module AbstractSet :
   functor (Elt : ORDERED_TYPE) ->
     sig
       type element = Elt.t
       type set
       val empty : set
       val add : element -> set -> set
       val member : element -> set -> bool
     end
```

Exercice (fonctions polymorphes et foncteurs)

- On veut définir une opération de tri polymorphe
  `tri : 'a list -> 'a list`
- On a besoin d'une opération de comparaison
  `lesseq: 'a -> a' -> bool`
- On est donc obligé de définir
  `tri : 'a list ->  ('a -> 'a -> bool) -> 'a list`

Exercice : Proposez une solution alternative selon le schéma suivant :

1. On définit une signature ORDTYPE 'type ordonné'.
2. On définit un foncteur qui prend en paramètre une structure 'type ordonné' et produit une structure avec une fonction de tri pour le type ordonné en question.

Une solution

```
(* Une signature pour les types ordonnés *)
module type ORDTYPE = sig
  type t
  val lesseq: t -> t -> bool
end ;;

(* Un module parametrique pour faire le tri par insertion *)
module Sort (OrdType : ORDTYPE ) = struct
  type t = OrdType.t
  let rec insert x l =  match l with
    [] -> [x] | y::l1 -> if OrdType.lesseq x y
                         then x::y::l1 else y:: insert x l1
  let rec isort l = match l with
    [] -> []  | y::l1 -> insert y  (isort l1)
end;;

(* Une structure de paires d'entiers avec ordre lexicographique *)
module OrdIntPair = struct
  type t = int * int
  let lesseq (x1,x2) (y1,y2) =
  if x1 <= y1 then true else (if x1=y1 then x2<= y2 else false)
end;;

(* Définition d'une structure Sort(OrdIntPair) *)
module S = Sort(OrdIntPair);;
```

```
(* Une liste de couples d'entiers *)

 # let l = [(2,4); (3,2); (1,5)];;
 val l : (int * int) list = [(2, 4); (3, 2); (1, 5)]

(* On utilise la structure pour trier la liste d'entiers *)

 # S.isort l;;
  - : OrdIntPair.t list = [(1, 5); (2, 4); (3, 2)]


(* La fonction insert est aussi visible *)

 # S.insert (4,5) (S.isort l);;
 - : OrdIntPair.t list = [(1, 5); (2, 4); (3, 2); (4, 5)] *)
```

```
(* Pour la cacher on definit *)

module Sort (OrdType : ORDTYPE ) = (
struct
  type t = OrdType.t
  let rec insert x l =  match l with
     [] -> [x] | y::l1 -> if OrdType.lesseq x y then x::y::l1
                          else y:: insert x l1
  let rec isort l = match l with
     [] -> []  | y::l1 -> insert y  (isort l1)
end
:
sig
  val isort : OrdType.t list -> OrdType.t list
end);;
```

Lectures conseillées

- J. Mitchell. Concepts in programming languages, chpt 9, Data abstraction and Modularity.

  Pour une perspective historique sur les notions d'abstraction et de modularité.

- E. Chailloux et al. Objective OCAML, chapitre 14, Programmation modulaire (en ligne).

  Pour les détails sur les modules en OCAML (ces transparents sont tirés de ce livre).

Lectures avancées

- D. Mac Queen. Modules for standard ML. ACM POPL, 1984.

  On décrit la conception du système de modules de ML.

- Claudio Russo. Recursive structures for standard ML ACM ICFP, 2001.
- D. Dreyer, K. Crary, R. Harper. A type system for higher-order modules. ACM POPL, 2003.

  We present a type theory for higher-order modules that accounts for many central issues in module system design, including translucency, applicativity, generativity, and modules as first-class values (...)

Classes vs. Modules

Outline

Outline

Complementary tools

Module system

The notion of *module* is taken seriously

- ⊕ Abstraction-based assembling language of structures
- ⊖ It does not help extensibility (unless it is by unrelated parts), does not love recursion

Class-based OOP

The notion of *extensibility* is taken seriously

- ⊕ Horizontally by adding new classes, vertically by inheritance
- ⊕ Value abstraction is obtained by hiding some components
- ⊖ Pretty rigid programming style, difficult to master because of late binding.

Modularity in OOP and ML

A three-layered framework

1. Interfaces
2. Classes
3. Objects

ML Modules

The intermediate layer (classes) is absent in ML module systems

This intermediate layer makes it possible to

1. Bind operations to instances
2. Specialize and redefine operations for new instances

Rationale

Objects can be seen as a generalization of “references” obtained by *tightly coupling* them with their operators

An example in Scala

```
trait Vector {
  def norm() : Double                               //declared method
  def isOrigin (): Boolean = (this.norm == 0)    // defined method
}
```

Like a Java interface but you can also give the definition of some methods.
When defining an instance of Vector I need only to specify `norm`:

```
class Point(a: Int, b: Int) extends Vector {
  var x: Int = a                        // mutable instance variable
  var y: Int = b                        // mutable instance variable
  def norm(): Double = sqrt(pow(x,2) + pow(y,2))         // method
  def erase(): Point = { x = 0; y = 0; return this }     // method
  def move(dx: Int): Point = new Point(x+dx,y)           // method
}
```

```
scala> new Point(1,1).isOrigin
res0: Boolean = false
```

Equivalently

```
class Point(a: Int, b: Int) {
  var x: Int = a                          // mutable instance variable
  var y: Int = b                          // mutable instance variable
  def norm(): Double = sqrt(pow(x,2) + pow(y,2))       // method
  def erase(): Point = { x = 0; y = 0; return this }    // method
  def move(dx: Int): Point = new Point(x+dx,y)          // method
  def isOrigin(): Boolean = (this.norm == 0)            // method
}
```

Equivalently? Not really:

```
class PolarPoint(norm: Double, theta: Double) extends Vector {
  var norm: Double = norm
  var theta: Double = theta
  def norm(): Double = return norm
  def erase(): PolarPoint = { norm = 0 ; return this }
}
```

Can use instances of both `PolarPoint` and `Point` (first definition but not the second) where an object of type `Vector` is expected.

Inheritance

```
class Point(a: Int, b: Int) {
  var x: Int = a
  var y: Int = b
  def norm(): Double = sqrt(pow(x,2) + pow(y,2))
  def erase(): Point = { x = 0; y = 0; return this }
  def move(dx: Int): Point = new Point(x+dx,y)
  def isOrigin(): Boolean = (this.norm == 0)
}

class ColPoint(u: Int, v: Int, c: String) extends Point(u, v) {
  val color: String = c           // non-mutable instance variable
  def isWhite(): Boolean = c == "white"
  override def norm(): Double = {
    if (this.isWhite) return 0 else return sqrt(pow(x,2)+pow(y,2))
  }
  override def move(dx: Int): ColPoint=new ColPoint(x+dx,y,"red")
}
```

`isWhite` added; `erase`, `isOrigin` inherited; `move`, `norm` overridden. Notice the late binding of `norm` in `isOrigin`.

Late binding of `norm`

```
scala> new ColPoint( 1, 1, "white").isOrigin
res1: Boolean = true
```

the method defined in Point is executed but `norm` is dynamically bound to the definition in `ColPoint`.

Role of each construction

Traits (interfaces): Traits are similar to *recursive record types* and make it possible to range on objects with common methods with compatible types but incompatible implementations.

```
type Vector = { norm: Double ,          // actually unit -> Double
                erase: Vector ,         // actually unit -> Vector
                isOrigin: Boolean       // actually unit -> Boolean
               }
```

Both `Point` and `PolarPoint` have the type above, but only if explicitly declared in the class (name subtyping: an explicit design choice to avoid unwanted interactions).

Classes: Classes are object templates in which instance variables are declared and the semantics of `this` is open (late binding).

Objects: Objects are instances of classes in which variables are given values and the semantic of `this` is bound to the object itself.

Late-binding and inheritance

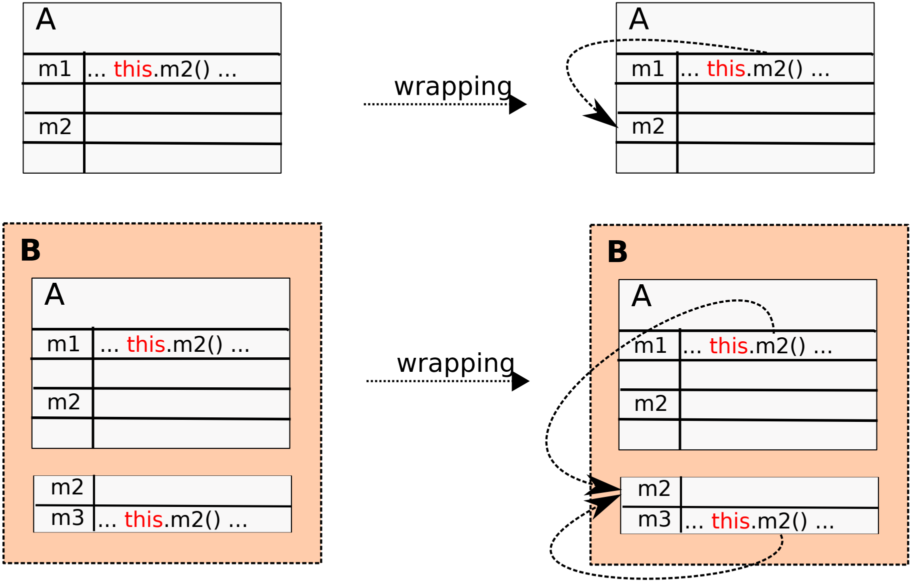
The tight link between objects and their methods is embodied by *late-binding*

Example

```
class A {
   def m1() = {.... this.m2() ...}
   def m2() = {...}
}

class B extends A {
   def m3() = {... this.m2() ...}
   override def m2() = {...}                //overriding
}
```

Two different behaviors according to whether late binding is used or not

Graphical representation



FP and OOP

- FP is a more operation-oriented style of programming
- OOP is a more state-oriented style of programming
- Modules and Classes+Interfaces are the respective tools for “programming in the large” and accounting for software evolution

Software evolution

Classes and modules are not necessary for small non evolving programs (except to support separate compilation)

They are significant for software that

- should remain extensible over time
  (e.g. add support for new target processor in a compiler)
- is intended as a framework or set of components to be (re)used in larger programs
  (e.g. libraries, toolkits)

Adapted to different kinds of extensions

Instances of programmer nightmares

- Try to modify the type-checking algorithm in the Java Compiler
- Try to add a new kind of account, (e.g. an equity portfolio account) to the example given for functors (see Example Chapter 14 OReilly book).

	FP approach	**OO approach**
Adding a new kind of things	Must edit all functions, by adding a new case to every pattern matching	Add one class (the other classes are unchanged)
Adding a new operation over things	Add one function (the other functions are unchanged)	Must edit all classes by adding or modifying methods in every class

Summary

Modules and classes play different roles:

- Modules handle type abstraction and parametric definitions of abstractions (functors)
- Classes do not provide this type abstraction possibility
- Classes provide late binding and inheritance (and message passing)

It is no shame to use both styles and combine them in order to have the possibilities of each one

Summary

Which one should I choose?

- *Any* of them when both are possible for the problem at issue
- *Classes* when you need late binding
- *Modules* if you need abstract types that share implementation (e.g. vectors and matrices)
- ***Both* in several cases.**

Trend

The frontier between modules and classes gets fussier and fuzzier

Not a clear-cut difference

- Mixin Composition
- Multiple dispatch languages
- OCaml Classes
- Haskell's type classes

Let us have a look to each point

Outline

Mixin Class Composition

Reuse the new member definitions of a class (i.e., the delta in relationship to the superclass) in the definition of a new class. In Scala:

```
abstract class AbsIterator {
  type T                              // opaque type as in OCaml Modules
  def hasNext: Boolean
  def next: T
}
```

Abstract class (as in Java we cannot instantiate it). Next define an interface (`trait` in Scala: unlike Java traits may specify the implementation of some methods; unlike abstract classes traits cannot interoperate with Java)

```
trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit) { while (hasNext) f(next) } // higher-order
}
```

A concrete iterator class, which returns successive characters of a string:

```
class StringIterator(s: String) extends AbsIterator {
  type T = Char
  private var i = 0
  def hasNext = i < s.length()
  def next = { val ch = s charAt i; i += 1; ch }
}
```

Cannot combine the functionality of StringIterator and RichIterator into a single class by single inheritance (as both classes contain member impementations with code). Mixin-class composition (keyword `with`): reuse the delta of a class definition (i.e., all new definitions that are not inherited)

```
object StringIteratorTest {
  def main(args: Array[String]) {
    class Iter extends StringIterator(args(0)) with RichIterator //mixin
    val iter = new Iter
    iter.foreach(println) }
}
```

Extends the “superclass” `StringIterator` with `RichIterator`’s methods that are not inherited from `AbsIterator`: `foreach` but not `next` or `hasNext`.

Note that the last application works since `println :  Any => Unit`:

```
 scala> def test (x : Any => Unit) = x     // works also if we replace
 test: ((Any) => Unit)(Any) => Unit       // Any by a different type

 scala> test(println)
 res0: (Any) => Unit = <function>
```

Rationale

Mixins are the “join” of an inheritance relation

Outline

Multiple dispatch languages

Originally used in functional languages

- The ancestor: CLOS (Common Lisp Object System)
- Cecil
- Dylan
- Now getting into mainstream languages by extensions (Ruby's `Multiple Dispatch` library, C# 4.0 dynamic or multi-method library, ...) or directly as in Perl 6.

Multiple dispatch in Perl 6

```
multi sub identify(Int $x) {
    return "$x is an integer."; }

multi sub identify(Str $x) {
    return qq<"$x" is a string.>; }   #qq stands for ‘‘double quote’’

multi sub identify(Int $x, Str $y) {
    return "You have an integer $x, and a string \"$y\"."; }

multi sub identify(Str $x, Int $y) {
    return "You have a string \"$x\", and an integer $y."; }

multi sub identify(Int $x, Int $y) {
    return "You have two integers $x and $y."; }

multi sub identify(Str $x, Str $y) {
    return "You have two strings \"$x\" and \"$y\"."; }

say identify(42);
say identify("This rules!");
say identify(42, "This rules!");
say identify("This rules!", 42);
say identify("This rules!", "I agree!");
say identify(42, 24);
```

Multiple dispatch in Perl 6

Embedded in classes

```
class Test {
    multi method identify(Int $x) {
      return "$x is an integer."; }
    }
    multi method identify(Str $x) {
      return qq<"$x" is a string.>;
    }
}
my Test $t .= new();
$t.identify(42);                # 42 is an integer
$t.identify("weasel");          # "weasel" is a string
```

Partial dispatch

```
multi sub write_to_file(str $filename , Int $mode ;; Str $text) {
   ...
}

multi sub write_to_file(str $filename ;; Str $text) {
   ...
}
```

Class methods as special case of partial dispatch

```
class Point {
    has $.x is rw;
    has $.y is rw;

    method set_coordinates($x, $y) {
        $.x = $x;
        $.y = $y;
    }
};

class Point3D is Point {
    has $.z is rw;

    method set_coordinates($x, $y) {
        $.x = $x;
        $.y = $y;
        $.z = 0;
    }
};

my $a = Point3D.new(x => 23, y => 42, z => 12);
say $a.x;                                        # 23
say $a.z;                                        # 12
$a.set_coordinates(10, 20);
say $a.z;                                        #  0
```

Equivalently with multi subroutines

```
class Point {
    has $.x is rw;
    has $.y is rw;
};

class Point3D is Point {
    has $.z is rw;
};

multi sub set_coordinates(Point $p ;; $x,  $y) {
    $p.x = $x;
    $p.y = $y;
};

multi sub set_coordinates(Point3D $p ;; $x,  $y) {
    $p.x = $x;
    $p.y = $y;
    $p.z = 0;
};

my $a = Point3D.new(x => 23, y => 42, z => 12);
say $a.x;                                         # 23
say $a.z;                                         # 12
set_coordinates($a, 10, 20);
say $a.z;                                         #  0
```

Nota Bene

There is no encapsulation here.

```
class Point {
    has $.x is rw;
    has $.y is rw;
};

class Point3D is Point {
    has $.z is rw;
};

multi sub set_coordinates(Point $p ;; $x,  $y) {
    $p.x = $x;
    $p.y = $y;
};

multi sub set_coordinates(Point3D $p ;; $x,  $y) {
    $p.x = $x;
    $p.y = $y;
    $p.z = 0;
};

my $a = Point3D.new(x => 23, y => 42, z => 12);
say $a.x;                                           # 23
say $a.z;                                           # 12
set_coordinates($a, 10, 20);
say $a.z;                                           #  0
```

Note this for the future (of the course)

```
class Point {
    has $.x is rw;
    has $.y is rw;
};

class Point3D is Point {
    has $.z is rw;
};

multi sub fancy(Point $p, Point3D $q) {
    say "first was called";
};

multi sub fancy(Point3D $p, Point $q) {
    say "second was called";
};

my $a = Point3D.new(x => 23, y => 42, z => 12);
fancy($a,$a)};
```

```
Ambiguous dispatch to multi 'fancy'. Ambiguous candidates had signatures:
:(Point $p, Point3D $q)
:(Point3D $p, Point $q)
in Main (file <unknown>, line <unknown>)
```

Outline

OCaml Classes

Some compromises are needed

- No polymorphic objects
- Need of explicit coercions
- ***No overloading*** **... Haskell makes exactly the opposite choice ...**

A brief parenthesis

A scratch course on OCaml classes and objects by Didier Remy (just click here) `http://gallium.inria.fr/~remy/poly/mot/2/index.html`

Programming is in general less liberal than in “pure” object-oriented languages, because of the constraints due to type inference.

Outline

Haskell’s Typeclasses

Typeclasses define a set of functions that can have different implementations depending on the type of data they are given.

```
class BasicEq a where
    isEqual :: a -> a -> Bool
```

An instance type of this typeclass is any type that implements the functions defined in the typeclass.

```
ghci> :type isEqual
isEqual :: (BasicEq a) => a -> a -> Bool
```

« For all types `a`, so long as `a` is an instance of `BasicEq`, `isEqual` takes two parameters of type `a` and returns a `Bool` »

To define an instance:

```
instance BasicEq Bool where
    isEqual True  True  = True
    isEqual False False = True
    isEqual _     _     = False
```

We can now use `isEqual` on `Bool`s, but not on any other type:

```
ghci> isEqual False False
True
ghci> isEqual False True
False
ghci> isEqual "Hi" "Hi"

<interactive>:1:0:
    No instance for (BasicEq [Char])
      arising from a use of ‘isEqual’ at <interactive>:1:0-16
    Possible fix: add an instance declaration for (BasicEq [Char])
    In the expression: isEqual "Hi" "Hi"
    In the definition of ‘it’: it = isEqual "Hi" "Hi"
```

As suggested we should add an instance for strings

```
instance BasicEq String where ....
```

A not-equal-to function might be useful. Here's what we might say to define a typeclass with two functions:

```
class BasicEq2 a where
    isEqual2 :: a -> a -> Bool
    isEqual2 x y = not (isNotEqual2 x y)

    isNotEqual2 :: a -> a -> Bool
    isNotEqual2 x y = not (isEqual2 x y)
```

People implementing this class must provide an implementation of at least one function. They can implement both if they wish, but they will not be required to.

Type-classes vs OOP

Type classes are like traits/interfaces/abstract classes, not classes itself (no *proper* inheritance and data fields).

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- let's just implement one function in terms of the other
  x /= y = not (x == y)
```

is, in a Java-like language:

```
interface Eq<A> {
  boolean equal(A x);
  boolean notEqual(A x) {        // default, can be overridden
     return !equal(x);
  }
}
```

Haskell typeclasses concern more overloading than inheritance. They are closer to multi-methods (overloading and no access control such as `private` fields), but only with *static dispatching*.

Type-classes vs OOP

A flavor of inheritance

They provide a very limited form of inheritance (but without overriding and late binding!):

```
class  Eq a => Ord a  where
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a
```

The *subclass* `Ord` “inherits” the operations from its *superclass* `Eq`. In particular, “methods” for subclass operations can assume the existence of “methods” for superclass operations:

```
class  Eq a => Ord a  where
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a
  x < y =  x <= y && x /= y
```

Inheritance thus is not on instances but rather on types (a Haskell class is not a type but a template for a type). *Multiple inheritance* is possible:

```
class (Real a, Fractional a) => RealFrac a where ...
```

Hybrid solutions

- Mixins raised in FP area (Common Lisp) and are used in OOP to allow minimal module composition (as functors do very well). On the other hand they could endow ML module system with inheritance and overriding
- Multi-methods are an operation centric version of OOP. They look much as a functional approach to objects
- OCaml and Haskell classes are an example of how functional language try to obtain the same kind of modularity as in OOP.

Something missing in OOP

What about Functors?

Outline

Generics in C#

Why in C# and not in Java?

Direct support in the CLR and IL (intermediate language)

The CLR implementation pushes support for generics into almost all feature areas, including serialization, remoting, reflection, reflection emit, profiling, debugging, and pre-compilation.

Java Generics based on GJ

Rather than extend the JVM with support for generics, the feature is "compiled away" by the Java compiler

Consequences:

- generic types can be instantiated only with reference types (e.g. string or object) and not with primitive types
- type information is not preserved at runtime, so objects with distinct source types such as List<string> and List<object> cannot be distinguished by run-time
- Clearer syntax

Generics Problem Statement

```
public class Stack
{
   object[] m_Items;
   public void Push(object item)
   {...}
   public object Pop()
   {...}
}
```

- runtime cost (boxing/unboxing, garbage collection)
- type safety

```
Stack stack = new Stack();
stack.Push(1);
stack.Push(2);
int number = (int)stack.Pop();

Stack stack = new Stack();
stack.Push(1);
string number = (string)stack.Pop();       // exception thrown
```

Heterogenous translation

You can overcome these two problems by writing type-specific stacks. For integers:

```
public class IntStack
{
   int[] m_Items;
   public void Push(int item){...}
   public int Pop(){...}
}
IntStack stack = new IntStack();
stack.Push(1);
int number = stack.Pop();
```

For strings:

```
public class StringStack
{
   string[] m_Items;
   public void Push(string item){...}
   public string Pop(){...}
}
StringStack stack = new StringStack();
stack.Push("1");
string number = stack.Pop();
```

Problem

Writing type-specific data structures is a tedious, repetitive, and error-prone task.

Solution

Generics

```
public class Stack<T>
{
   T[] m_Items;
   public void Push(T item)
   {...}
   public T Pop()
   {...}
}
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
int number = stack.Pop();
```

You have to instruct the compiler which type to use instead of the generic type parameter T, both when declaring the variable and when instantiating it:

`Stack<int> stack = new Stack<int>();`

```
public class Stack<T>{
   readonly int m_Size;
   int m_StackPointer = 0;
   T[] m_Items;
   public Stack():this(100){
   }
   public Stack(int size){
      m_Size = size;
      m_Items = new T[m_Size];
   }
   public void Push(T item){
      if(m_StackPointer >= m_Size)
         throw new StackOverflowException();
      m_Items[m_StackPointer] = item;
      m_StackPointer++;
   }
   public T Pop(){
      m_StackPointer--;
      if(m_StackPointer >= 0) {
         return m_Items[m_StackPointer]; }
      else {
         m_StackPointer = 0;
         throw new InvalidOperationException("Cannot pop an empty stack");
      }
   }
}
```

Recap

Two different styles to implement generics (when not provided by the VM):

1. *Homogenous:* replace occurrences of the type parameter by the type `Object`. This is done in GJ and, thus, in Java (>1.5).
2. *Heterogeneous:* make one copy of the class for each instantiation of the type parameter. This is done by C++ and Ada.

The right solution is to support generics directly in the VM

Unfortunately, Javasoft marketing people did not let Javasoft researchers to change the JVM.

Multiple Generic Type Parameters

```
class Node<K,T> {
   public K Key;
   public T Item;
   public Node<K,T> NextNode;
   public Node() {
      Key      = default(K);          // the "default" value of type K
      Item     = default(T);          // the "default" value of type T
      NextNode = null;
   }
   public Node(K key,T item,Node<K,T> nextNode) {
      Key      = key;
      Item     = item;
      NextNode = nextNode;
   }
}

public class LinkedList<K,T> {
   Node<K,T> m_Head;
   public LinkedList() {
      m_Head = new Node<K,T>();
   }
   public void AddHead(K key,T item){
      Node<K,T> newNode = new Node<K,T>(key,item,m_Head);
      m_Head = newNode;
   }
}
```

Generic Type Constraints

Suppose you would like to add searching by key to the linked list class

```
public class LinkedList<K,T> {

   public T Find(K key) {
      Node<K,T> current = m_Head;
      while(current.NextNode != null) {
         if(current.Key == key)                     //Will not compile
            break;
         else
            current = current.NextNode;
      }
      return current.Item;
   }
   // rest of the implementation
}
```

The compiler will refuse to compile this line
`if(current.Key == key)`

because the compiler does not know whether K (or the actual type supplied by the client) supports the == operator.

We must ensure that K implements the following interface

```
public interface IComparable  {
   int CompareTo(Object other);
   bool Equals(Object other);
}
```

This can be done by specifying a constraint:

```
public class LinkedList<K,T> where K : IComparable {
   public T Find(K key) {
      Node<K,T> current = m_Head;
      while(current.NextNode != null) {
         if(current.Key.CompareTo(key) == 0)
            break;
         else
            current = current.NextNode;
      }
      return current.Item;
   }
   //Rest of the implementation
}
```

Problems

1. `key` is boxed/unboxed when it is a value (i.e. not an object)
2. The static information that `key` is of type `K` is not used (`CompareTo` requires a parameter just of type `Object`).

F-bounded polymorphism

In order to enhance type-safety (in particular, enforce the argument of `K.CompareTo` to have type `K` rather than `Object`) and avoid boxing/unboxing when the key is a value, we can use a generic version of IComparable.

```
public interface IComparable<T> {
   int CompareTo(T other);
   bool Equals(T other);
}
```

This can be done by specifying a constraint:

```
public class LinkedList<K,T> where K : IComparable<K> {
   public T Find(K key) {
      Node<K,T> current = m_Head;
      while(current.NextNode != null) {
         if(current.Key.CompareTo(key) == 0)
            break;
         else
            current = current.NextNode;
      }
      return current.Item;
   }
   //Rest of the implementation
}
```

Generic methods

You can define method-specific (possibly constrained) generic type parameters even if the containing class does not use generics at all:

```
public class MyClass
{
   public void MyMethod<T>(T t) where T : IComparable<T>

   {...}
}
```

When calling a method that defines generic type parameters, you can provide the type to use at the call site:

```
MyClass obj = new MyClass();
obj.MyMethod<int>(3)
```

Subtyping

Generics are *invariant*:

```
List<string> ls = new List<string>();
ls.Add("test");
List<object> lo = ls;   // Can't do this in C#
object o1 = lo[0];      // ok - converting string to object
lo[0] = new object();   // ERROR - can't convert object to string
```

This is the right decision as the example above shows.
Thus

S is a subtype of T *does not imply* Class<S> is a subtype of Class<T>.

If this (covariance) were allowed, the last line would have to result in an exception (eg. InvalidCastException).

Beware of self-proclaimed type-safety

Since `S` is a subtype of `T` *implies* `S[]` is subtype of `T[]`. (*covariance*)

Do not we have the same problem with arrays? **Yes**

From Jim Miller CLI book

> *The decision to support covariant arrays was primarily to allow Java to run on the VES (Virtual Execution System). The covariant design is not thought to be the best design in general, but it was chosen in the interest of broad reach.*

(yes, it is not a typo, Microsoft decided to break type safety and did so in order to run Java in `.net`*)*

Regretful (and regretted) decision:

```
class Test {
   static void Fill(object[] array, int index, int count, object val) {
      for (int i = index; i < index + count; i++) array[i] = val;
   }
   static void Main() {
      string[] strings = new string[100];
      Fill(strings, 0, 100, "Undefined");
      Fill(strings, 0, 10, null);
      Fill(strings, 90, 10, 0);    //→System.ArrayTypeMismatchException
   }
}
```

Variant annotations

Add variants (C# 4.0)

```
// Covariant parameters can be used as result types
interface IEnumerator<out T> {
      T Current { get; }
      bool MoveNext();
}
// Covariant parameters can be used in covariant result types
interface IEnumerable<out T> {
      IEnumerator<T> GetEnumerator();
}
// Contravariant parameters can be used as argument types
interface IComparer<in T> {
      bool Compare(T x, T y);
}
```

This means we can write code like the following:

```
IEnumerable<string> stringCollection = ...;                //smaller type
IEnumerable<object> objectCollection = stringCollection; //larger type
foreach( object o in objectCollection ) { ... }

IComparer<object> objectComparer = ...;                     //smaller type
IComparer<string> stringComparer = objectComparer;         //larger type
bool b = stringComparer.Compare( "x", "y" );
```

Features becoming standard in modern OOLs . . .

In Scala we have generics classes and methods with annotations and bounds

```
class ListNode[+T](h: T, t: ListNode[T]) {
  def head: T = h
  def tail: ListNode[T] = t
  def prepend[U >: T](elem: U): ListNode[U] =
    ListNode(elem, this)
}
```

and F-bounded polymorphism as well:

```
class GenCell[T](init: T) {
  private var value: T = init
  def get: T = value
  def set(x: T): unit = { value = x }
}

trait Ordered[T] {
  def < (x: T): boolean
}

def updateMax[T <: Ordered[T]](c: GenCell[T], x: T) =
  if (c.get < x) c.set(x)
```

… but also in FP.

All these characteristics are present in different flavours in OCaml

Generics are close to parametrized classes:

```
# exception Empty;;

class ['a] stack =
  object
   val mutable p : 'a list =  []
   method push x = p <- x :: p
   method pop =
     match p with
     | [] -> raise Empty
     | x::t -> p <- t; x
  end;;
class ['a] stack :
object val mutable p : 'a list method pop : 'a method push : 'a -> unit en
# new stack # push 3;;
- : unit = ()
# let x = new stack;;
val x : '_a stack = <obj>
# x # push 3;;
- : unit = ()
# x;;
- : int stack = <obj>
```

Constraints can be deduced by the type-checker

```
#class ['a] circle (c : 'a) =
   object
     val mutable center = c
     method center = center
     method set_center c = center <- c
     method move = (center#move : int -> unit)
   end;;
class ['a] circle :
  'a ->
  object
    constraint 'a = < move : int -> unit; .. >
    val mutable center : 'a
    method center : 'a
    method move : int -> unit
    method set_center : 'a -> unit
  end
```

Constraints can be imposed by the programmer

```
#class point x_init =
   object
     val mutable x = x_init
     method get_x = x
     method move d = x <- x + d
   end;;
class point :
  int ->
  object val mutable x : int method get_x : int method move : int -> unit e

#class ['a] circle (c : 'a) =
   object
     constraint 'a = #point   (* = < get_x : int; move : int->unit; .. > *)
     val mutable center = c
     method center = center
     method set_center c = center <- c
     method move = center#move
   end;;
class ['a] circle :
  'a ->
  object
    constraint 'a = #point
    val mutable center : 'a
    method center : 'a
    method move : int -> unit
    method set_center : 'a -> unit
  end
```

Explicit instantiation is done just for inheritance

```
#class colored_point x (c : string) =
   object
     inherit point x
     val c = c
     method color = c
   end;;
class colored_point :
  int ->
  string ->
  object
   :
   :
  end

#class colored_circle c =
   object
     inherit [colored_point] circle c
     method color = center#color
   end;;
class colored_circle :
  colored_point ->
  object
    val mutable center : colored_point
    method center : colored_point
    method color : string
    method move : int -> unit
    method set_center : colored_point -> unit
  end
```

Variance constraints

- Variance constraint are meaningful only with subtyping (i.e. objects, polymorphic variants, . . .).
- They can be used in OCaml (not well documented): useful on abstract types to describe the expected behaviour of the type with respect to subtyping.
- For instance, an immutable container type (like lists) will have a covariant type:

  ```
  type (+'a) container
  ```

  meaning that if s is a subtype of t then s container is a subtype of t container. On the other hand an acceptor will have a contravariant type:

  ```
  type (-'a) acceptor
  ```

  meaning that if s is a subtype of t then t acceptor is a subtype s acceptor.

see also `https://ocaml.janestreet.com/?q=node/99`

Summary for generics . . .

Generics endow OOP with features from the FP universe

Generics on classes (in particular combined with Bounded Polymorphism) look close to functors.

Compare the Scala program in two slides with the `Set` functor with signature:

```
module Set :
functor (Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      type set = element list
      val empty : 'a list
      val add : Elt.t -> Elt.t list -> Elt.t list
      val member : Elt.t -> Elt.t list -> bool
    end
```

where

```
type comparison = Less | Equal | Greater;;

module type ORDERED_TYPE =
    sig
     type t
     val compare: t -> t -> comparison
    end;;
```

and that is defined as:

```
module Set (Elt: ORDERED_TYPE) =
  struct
    type element = Elt.t
    type set = element list
    let empty = []
    let rec add x s =
      match s with
        [] -> [x]
      | hd::tl ->
         match Elt.compare x hd with
           Equal   -> s          (* x is already in s *)
          | Less    -> x :: s     (* x is smaller than all elmts of s *)
          | Greater -> hd :: add x tl
    let rec member x s =
      match s with
        [] -> false
      | hd::tl ->
          match Elt.compare x hd with
            Equal   -> true      (* x belongs to s *)
           | Less    -> false     (* x is smaller than all elmts of s *)
           | Greater -> member x tl
  end;;
```

```
trait Ordered[A] {
  def compare(that: A): Int
  def < (that: A): Boolean =  (this compare that) <  0
  def > (that: A): Boolean =  (this compare that) >  0
}

trait Set[A <: Ordered[A]] {
  def add(x: A): Set[A]
  def member(x: A): Boolean
}

class EmptySet[A <: Ordered[A]] extends Set[A] {
  def member(x: A): Boolean = false
  def add(x: A): Set[A] =
             new NonEmptySet(x, new EmptySet[A], new EmptySet[A])
}

class NonEmptySet[A <: Ordered[A]]
      (elem: A, left: Set[A], right: Set[A]) extends Set[A] {
  def member(x: A): Boolean =
     if (x < elem) left member x
     else if (x > elem) right member x
     else true
  def add(x: A): Set[A] =
     if (x < elem) new NonEmptySet(elem, left add x, right)
     else if (x > elem) new NonEmptySet(elem, left, right add x)
     else this
}
```

Generics endow OOP with features from the FP universe

Generics on methods bring the advantages of parametric polymorphism

```
def isPrefix[A](p: Stack[A], s: Stack[A]): Boolean = {
  p.isEmpty ||
  p.top == s.top && isPrefix[A](p.pop, s.pop)
}

val s1 = new EmptyStack[String].push("abc")
val s2 = new EmptyStack[String].push("abx").push(s1.top)
println(isPrefix[String](s1, s2))
```

Local Type Inference

It is possible to deduce the type parameter from `s1` and `s2`. Scala does it for us.

```
val s1 = new EmptyStack[String].push("abc")
val s2 = new EmptyStack[String].push("abx").push(s1.top)
println(isPrefix(s1, s2))
```

Computational effects

Outline

Programmation Fonctionnelle non pure

Même si OCaml est un langage fonctionnel, nous avons souvent utilisé dans les exemples passés des caracteristiques qui sortent d'une programmation fonctionnelle pure. C'est le cas de :

- exceptions,
- operations d'entrée/sortie,
- références,
- continuations explicites.

Nous allons les étudier dans cette partie

Outline

Exception

Typage et domaine de définition

type inféré $\neq$ domaine de définition

- Le typage est une approximation
- Exemples : division entière, tête de liste vide
- provient souvent d'un filtrage non exhaustif

Que faire ?

- utiliser une valeur spéciale

```
# asin 2.;;
- : float = nan                  (* not-a-number IEEE standard *)
```

- effectuer une rupture de calcul jusqu'à un récupérateur

```
# 3/0;;
Exception: Division_by_zero.
```

exceptions

Syntaxe

```
exception E
```

ou

```
exception E of t;;
```

- une exception est une valeur de type `exn`
- le type `exn` est un type somme monomorphe *extensible*

```
# exception A_MOI;;
exception A_MOI

# A_MOI;;
- : exn = A_MOI

# exception Depth of int;;
exception Depth of int

# Depth 4;;
- : exn = Depth(4)

# exception Value of 'a ;;        (* monomorphe *)
Error: Unbound type parameter 'a
```

Déclaration et déclenchement d'une exception (1)

Déclenchement d'une exception

```
# raise;;
- : exn -> 'a = <fun>
```

- impossible à écrire (primitive)
- l'expression `raise` *E* n'a pas de contrainte de type

```
# raise A_MOI;;
Uncaught exception: A_MOI

# let x = 18;;
val x : int = 18

# if (x = 0) then raise A_MOI else x;; (* A_MOI used in int position *
- : int = 18
```

Déclaration et déclenchement d’une exception (2)

Avec un paramètre

```
# exception Echec of string;;
exception Echec of string

# let declenche_echec s = raise (Echec s);;
val declenche_echec : string -> 'a = <fun>

# declenche_echec "argument invalide";;
Exception: Echec "argument invalide".

# failwith;;                        (* ≃ fun x -> raise (Failure x) *)
- : string -> 'a = <fun>
```

Déclaration et déclenchement d’une exception (3)

Filtrage de motifs incomplet : déclanchement “involontaire”....

```
# let tete l = match l with t::q -> t;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val tete : 'a list -> 'a = <fun>

# tete [1;2;3];;
- : int = 1

# tete [];;
Exception: Match_failure ("", 13, 35).
```

.... ou volontaire afin d’interrompre une execution

```
# exception Found_zero;;
exception Found_zero

# let rec mult_aux l= match l with
    h::[] -> h
  | 0::t -> raise Found_zero
  | h::t -> h * mult_aux t;;
Warning: this pattern-matching ...
val mult_aux : int list -> int = <fun>
```

Récupération d'exceptions

Syntaxe :

try *expr* with *filtrage*

Le type des motifs du filtrage doit être exn.

```
# let mult_list l = match l with
   [] -> 0
| lo -> try mult_aux lo with Found_zero -> 0;;
val mult_list : int list -> int = <fun>

# mult_list [1;2;3;0;5;6];;
- : int = 0
```

Utilisation des exceptions

1. Gestion de situations exceptionnelles où le calcul ne peut pas se poursuivre $\rightarrow$ rupture du calcul
2. style de programmation (par example, rupture d'une boucle)

Attention au coût du `try` qui doit sauver le context courant
(à placer le plus exterieurement possible surtout s'il y a des boucles)

Exercice

Filtrage d'une liste

- filtrage des éléments d'une liste par un prédicat
- sans recopie inutile

On veut donc une function `filter` qui prend un predicat `p: 'a -> Bool` et une liste `lst` et dont l'implementation

1. Retourne immediatement `lst` si tous les élement de `lst` satisfont `p`
2. Si la liste est `h::t` et `h` satisfait `p`, elle retourne `h::(filter p t)`, et `(filter p t)` si `h` ne satisfait pas `p`

Utiliser les exceptions pour obtenir le comportement voulu

Une solution

```
# exception Identity;;
exception Identity

# let filter p l =
    let apply f x = try f x with Identity -> x in
    let rec fil l = match l with
      | [] -> raise Identity
      | h :: t -> if p h then      (* depuis la dernière application de *)
                    h :: fil t     (* apply tout element satisfait p    *)
                  else
                    apply fil t in (* on reccommence par un apply       *)
   apply fil l;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

La fonction `fil` fait une copie de la queue de la liste qui satisfait la propriété, mais cette copie est “garbage collectable” à la fin de l’appel de `filter`.

```
# let l1 = ....            (* liste avec un important memory footprint *)
# let l2 = filter p l1
```

si tout élément de `l1` satisfait `p` alors `l1` et `l2` dénoteront “physiquement" la même liste en mémoire (c’est équivalent à “`let l1 = l2`”). Plus en général `l1` et `l2` partageront en mémoire toute portion terminale de `l1` qui satisfait `p`.

Outline

Traits impératifs

Modèle plus proche de la couche physique

- Un bon condensé de programmation impérative :

```
x := x + 1
```

  - exécution d'une instruction (action) qui modifie l'état mémoire
  - passage à une nouvelle instruction dans le nouvel état mémoire
- modèle des langages Fortan, Pascal, C, Ada, . . .

Entrée/Sorties

- types : `in_channel` ou `out_channel`.

```
# open_in;;
- : string -> in_channel = <fun>

# open_out;;
- : string -> out_channel = <fun>

# close_in ;;
- : in_channel -> unit = <fun>

# close_out ;;
- : out_channel -> unit = <fun>
```

- exception : `End_of_file`
- canaux prédéfinis : `stdin`, `stdout` et `stderr`
- type `open_flag` pour les modes d'ouverture

Principales fonctions de I/O

```
input          :  in_channel → string → int → int → int (ch buf pos lgth
input_line     :  in_channel → string
output         :  out_channel → string → int → int → unit
output_string  :  out_channel → string → unit
read_line      :  unit → string
read_int       :  unit → int
print_string   :  string → unit
print_int      :  int → unit
print_newline  :  unit → unit
```

Example : “c’est plus/c’est moins”

```
# let rec cpcm n =
    let () = print_string "taper un nombre : " in
     let i = read_int ()
     in
       if i = n then
         let () = print_string "BRAVO" in
         let () = print_newline ()
         in print_newline ()
       else
         let () =
           if i < n then
             let () = print_string "C+"
             in print_newline ()
           else
             let () = print_string "C-"
             in print_newline ()
         in cpcm n ;;
val cpcm : int -> unit = <fun>

# cpcm 64;;
taper un nombre : 88
C-
taper un nombre : 44
C+
taper un nombre : 64
BRAVO
```

Valeurs physiquement modifiables

- valeurs structurées dont une partie peut être physiquement (en mémoire) modifiée ;
- vecteurs, enregistrements à champs modifiables, chaînes de caractères, références

Attention

Nécessite de contrôler l'ordre du calcul (mais les I/O aussi)

Ça tombe bien : OCaml est *strict*

Une fonction est *stricte* si lorsqu'elle est appliquée à un argument qui ne termine pas, elle ne termine pas. Un langage est *strict* s'il ne peut définir que des fonctions strictes. Un langage avec "eager evaluation" (les variables ne sont liés qu'à des valeurs) est toujours strict.

Vecteurs

- regroupent un nombre connu d'éléments de même type
- création : `Array.create : int → 'a → 'a array`,
- longueur : `Array.length : 'a array→ int`
- accès : `e1.(e2)`
- modification : `e1.(e2)<-e3`

```
# let v = Array.create 4 3.14;;
val v : float array = [|3.14; 3.14; 3.14; 3.14|]

# v.(1);;
- : float = 3.14

# v.(8);;
Exception: Invalid_argument "Array.get".

# v.(0) <- 100.;;
- : unit = ()

# v;;
- : float array = [|100.; 3.14; 3.14; 3.14|]
```

Vecteurs

Fonctions sur les vecteurs

- création matrice :
  - `Array.make_matrix : int → int → 'a → 'a array array`
- itérateurs :
  - `iter : ('a → unit) → 'a array → unit`
  - `map : ('a → 'b) → 'a array → 'b array`
  - `iteri : (int → 'a → unit) → 'a array → unit`
    (argument supplementaire = index de l'element dans l'array)
  - `mapi, fold_left, fold_right, . . .`

Enregistrements à champs mutables

Dans un enregistrement OCaml il est possible de specifier des champs qui sont modifiables.

- indication à la déclaration de type d'un champs est “mutable”
- accès au champ identique `e.f`
- modification similaire aux vecteurs `e1.f<-e2`

```
# type point = {mutable x : float; mutable y : float};;
type point = { mutable x: float; mutable y: float }

# let p = {x=1.; y=1.};;
val p : point = {x=1; y=1}

# p.x <- p.x +. 1.0;;
- : unit = ()

# p;;
- : point = {x=2; y=1}
```

Chaînes de caractères

- les chaînes sont des valeurs modifiables (fonction input)
- accès : `e1.[e2]`
- modification : `e1.[e2 ]<-e3`

```
# let s = "bonjour";;
val s : string = "bonjour"

# s.[3];;
- : char = 'j'

# s.[3]<-'t';;
- : unit = ()

# s;;
- : string = "bontour"
```

(N.B. Since Ocaml 4.02 this is no longer possible. This version introduces a new Bytes module for mutable strings while String becomes for data structures that cannot be modified in place)

Références

On prefere l'utilisation des records mutables, par lesquels ils sont desormais encodés

- type : `'a ref` (≡ `{mutable contents: 'a}`)
- read : `!e` (≡ `e.contents`)
- write : `e1:=e2` (≡ `e1.contents<-e2`)

```
# let incr x = x := !x + 1;;                    (* fonction prédefinie *)
val incr : int ref -> unit = <fun>

# let z = ref 3;;
val z : int ref = {contents=3}              (* noter le mutable record *)

# incr z;;
- : unit = ()

# z;;
- : int ref = {contents=4}

# (ref 3) := 2;;
- : unit = ()
```

Structures de contrôle

- composition séquentielle : `e1; e2`
  il ne s'agit que de sucre syntaxique pour : `let _ = e1 in e2`
  le type de la séquence est le type de `e2`
  - `e1` doit être de type `unit`
  - Si `e1` n'est pas de type `unit` cela cause un Warning :

    ```
    # 1;();;
    Warning S: this expression should have type unit.
    - : unit = ()
    # ignore;;
    - : 'a -> unit = <fun>
    # ignore 1;();;
    - : unit = ()
    ```
- conditionnelle : `if c then e` (où `e` est de type `unit`)
- itératives :
  - `while c do e done`
  - `for x=e1 [down]to e2 do e3 done`

La conditionnelle et les boucles sont des expressions de type `unit`

Exemple : somme de 2 vecteurs

```
#let somme a b =
  let al = Array.length a and bl = Array.length b in
  if al <> bl then failwith "somme"
  else if al = 0 then a
    else
      let c = Array.create al a.(0) in
         for i=0 to al-1 do
           c.(i) <- a.(i) + b.(i)
         done;
         c;;
val somme : int array -> int array -> int array = <fun>

# somme [|1; 2; 3|] [| 9; 10; 11|];;
- : int array = [|10; 12; 14|]
```

Style fonctionnel ou impératif ?

Utiliser le bon style selon les structures de données et leurs manipulations (par copie ou en place)

- impératif sur les matrices (en place)
- fonctionnel sur les arbres (par copie)

Mélanger les deux styles

- valeurs fonctionnelles modifiables
- implantation de l'évaluation retardée

Example 1 : Map

En style fonctionnel :

```
# let rec fmap f = function
  | [] -> []
  | h::t -> (f h)::(fmap f t);;
  val fmap : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

En style imperatif :

```
 # let imap f l =
    let nl = ref l
    and nr = ref [] in
      while (!nl <> []) do
        nr := ( f (List.hd !nl)) :: (!nr);
        nl := List.tl !nl
      done;
      List.rev !nr;;
 val imap : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Functional wins !

Exercise : écrire `fmap` en style tail-recursive

Example 2 : Transposée de matrice

En style imperatif :

```
# let itrans m =
    let l = Array.length m in
      for i=0 to l-1 do
         for j=i to l-1 do
           let v = m.(i).(j) in
             m.(i).(j) <- m.(j).(i);
             m.(j).(i) <- v
         done
      done;;
val itrans : 'a array array -> unit = <fun>
```

En style fonctionnel :

```
# let rec ftransl  = function
    | []::_ -> []
    |    l  -> (List.map List.hd l) :: ftransl (List.map List.tl l);;
val ftransl : 'a list list -> 'a list list = <fun>
```

Imperative wins !

Example 3 : Simulation d’évaluation paresseuse

Le calcul est gélé dans un thunk

```
# type 'a thunk = Exp of (unit -> 'a) | Val of 'a;;
type 'a thunk = Exp of (unit -> 'a) | Val of 'a

# type 'a delayed = {mutable thunk : 'a thunk};;
type 'a delayed = { mutable thunk : 'a thunk; }

# let delay f = { thunk = Exp f };;
val delay : (unit -> 'a) -> 'a delayed = <fun>

# let force e = match e.thunk with
         Val v -> v
      |  Exp f -> let v = f() in (e.thunk <- Val v ; v);;
val force : 'a delayed -> 'a = <fun>
```

- Pour retarder l’evaluation de `e` il faut écrire `delay(fun()-> e)` (qui a le type `t delayed` si `e` est de type `t`
- Pour forcer l’évaluation de `e` il faut écrire `force e` (où `e` est de type `t delayed`)
- Toute expression gélee ne sera évaluée qu’une seule fois.

Example d'évaluation paresseuse

```
# let test1 x = (print_string "pong" ;
                 print_newline();
                 x+x) in
  let arg = (print_string "ping";
             print_newline();
             6*8) in
  test1 arg;;
ping
pong
- : int = 96


# let test2 x = (print_string "pong" ;
                 print_newline();
                 force x + force x) in
  let arg = delay ( fun () -> print_string "ping";
                              print_newline();
                              6*8) in
  test2 arg;;
pong
ping
- : int = 96
```

Winning mix

Module `Lazy`

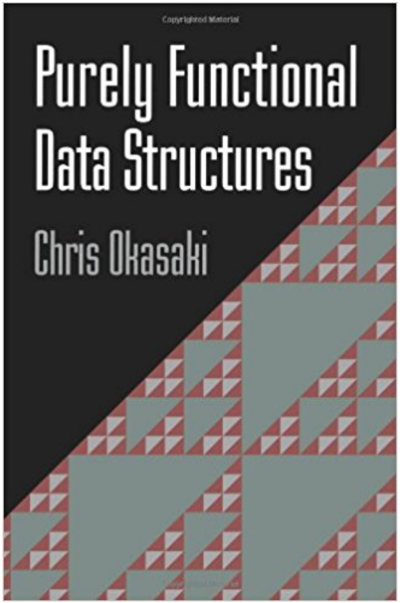
En Ocaml force et delay sont respectivement `Lazy.force` et `lazy` :

```
# let test2 x = ( print_string "pong" ;
                  print_newline();
                  Lazy.force x + Lazy.force x) in
  let arg = lazy( print_string "ping";
                  print_newline();
                  6*8) in
  test2 arg;;
pong
ping
- : int = 96
```

Problèmes

- Beaucoup moins éfficace qu'une implémentation native tel que Haskell
- Il n'a pas tous les avantages d'un langage non-strict (e.g. déforestation)
- Même s'il permet la définition de structures paresseuses, il n'a pas la même flexibilité qu'une implantation native.
  (e.g. `test2: int Lazy.t -> int` ... on ne peut pas lui passer un `int`).

Une bonne lecture



Outline

Opérateur `call/cc`

La fonction `call/cc` est un opérateur de contrôle qui capture la continuation courante et l'applique à son argument.
Obscur ?
Nous allons l'expliquer en détail lors des transformations de programmes.
Pour l'instant il suffit de savoir qu'elle est implantée en OCaml mais comme dit son auteur (Xavier Leroy) :

> *This library implements the call/cc (call-with-current-continuation) control operator for Objective Caml. This is a very naive implementation : it works only in bytecode, and performance is terrible (call/cc copies the whole stack). It is intended for educational and experimental purposes. Use in production code is not advised.*

Program transformations

Outline

Outline

The fuss about purity

High level features (stores, exceptions, I/O, . . .) are essential:

- A program execution has a *raison d'être* only if it has I/O
- Processors have registers not functions
- Databases store persistent data
- Efficiency and clarity can be more easily achieved with exceptions

Question

Why some widespread languages, such as Haskell, insists on *purity*?

Advantages of a pure functional framework

- Much easier static analysis and correctness proofs
- Lazy evaluation
- Program optimizations

Static analysis: some examples

- Dependence analysis:
  - control dependencies: the evaluation of a program's expressions depends on the result of a previous expression (eg, if_then_else)
  - data dependencies: the result of a program's expressions depends on the result of a previous expression (eg, let-expression)

  Dependence analysis determines whether or not it is safe to reorder or parallelize the evaluation of expressions.
- Data-flow analysis:
  - reaching definitions: determines which definitions may reach a given point in the code (eg, registers allocation)
  - live variable analysis: calculate for each program point the variables that may be potentially read (eg, use for dead-code elimination)

  Data-flow analysis gathers information about the possible set of values calculated at various points.
- Type systems

Exercises

1. Try to imagine why the presence of impure expressions can make both dependence and data-flow analysis more difficult
2. Try to think about the problems of implementing a static type system to ensure that there won't be any uncaught exception.

Check also Appel's book

Lazy evaluation (1)

In lazy (as opposed to strict/eager) evaluation an expression passed as argument:

- is only evaluated if the result is required by the calling function (delayed evaluation)
- is only evaluated to the extent that is required by the calling function, called (short-circuit evaluation).
- is never evaluated more than once (as in applicative-order evaluation)

Example

$$(\lambda x.(\mathrm{fst}\ x, \mathrm{fst}\ x))((\lambda y.(3+y,e))5)$$

$$\begin{array}{ll}
\to & (\mathrm{fst}\ ((\lambda y.(3+y,e))5), \mathrm{fst}\ ((\lambda y.(3+y,e))5))) \\
\to & (\mathrm{fst}\ (3+5,e), \mathrm{fst}\ (3+5,e)) \\
\to & (3+5, 3+5)
\end{array}$$

(The last reduction is an optimization: common subexpressions elimination)

Lazy evaluation (2)

In OCaml lazy evaluation can be implemented by *memoization*:

```
# let rec boucle = function 0 -> () | n -> boucle (n-1);;
val boucle : int -> unit = <fun>
# let gros_calcul () = boucle 100000000; 4;;
val gros_calcul : unit -> int = <fun>
# let v = gros_calcul ();;      (* it is slow *)
val v : int = 4
# v + 1;;                       (* it is fast *)
- : int = 5
#  let v () = gros_calcul ();; (* it is fast *)
val v : unit -> int = <fun>
# v () + 1;;                    (* it is slow *)
- : int = 5
# v () + 1;;                    (* it is slow *)
- : int = 5
# let v =
    let r = ref None in
      fun () -> match !r with
              | Some v -> v
              | None -> let v = (gros_calcul ()) in r := Some v; v;;
val v : unit -> int = <fun>
# v () + 1;;                     (* it is slow *)
- : int = 5
# v () + 1;;                     (* it is fast *)
- : int = 5
```

The Lazy module in OCaml

This is so frequent that OCaml provides this behavior natively via the special syntax `lazy` and the module `Lazy`:

```
# let v = lazy (gros_calcul ());;
val v : int lazy_t = <lazy>

# Lazy.force v;;                   (* it is slow *)
- : int = 4

# Lazy.force v;;                   (* it is fast *)
- : int = 4
```

Lazy evaluation (3)

Advantages

- Lazy data structures: possibly infinite, efficient copy, low memory footprint
- Better performance due to avoiding unnecessary calculations (?),
- Maintains purity (!)

Rationale

Since also strict languages can be endowed with laziness (see `Lazy` library in OCaml) then the clear advantage of *pervasive* lazy evaluation is to keep purity and, thus, referential transparency (not the other way round).

Optimizations

Purity makes important optimizations possible

1. Obvious program transformations. In Haskell

   ```
   map f (map g lst) = map (f.g) lst
   ```

   What if f and g had side effects?
   This is called “deforestation” and works for non-strict languages (in strict languages it may transform a function that does not terminates into one that terminates).
2. Function inlining, partial evaluation
3. Memoization
4. Common subexpressions elimination
5. Parallelization
6. Speculative evaluation
7. Other optimizations (see CPS part later on)

Program transformations

Previous optimizations are implemented by *program transformations*.

Meaning:

In the broadest sense: all translations between programming languages that preserve the meaning of programs.

Usage:

Typically used as passes in a compiler. Progressively bridge the gap between high-level source languages and machine code.

In this course:

We focus on translations between different languages. Translations within the same language are for optimization and studied in compiler courses.

The interest is twofold:

1. Eliminate high-level features of a language and target a smaller or lower-level language.
2. To program in languages that lack a desired feature. E.g. use higher-order functions or objects in C; use imperative programming in Haskell or Coq.

Transformations

Considered transformations

We will show how to get rid of higher level features:

- High-order functions
- “Impure” features: exceptions, state, call/cc

Note

In order to simulate higher level features we first have to formally define their semantics.

Let us take a refresher course on operational semantics and reduction strategies

Outline

Syntax and small-step semantics

Syntax

$$
\begin{array}{llcll}
\textit{Terms} & a, b & ::= & N & \text{Numeric constant} \\
 & & \mid & x & \text{Variable} \\
 & & \mid & a\,b & \text{Application} \\
 & & \mid & \lambda x.a & \text{Abstraction} \\
\textit{Values} & v & ::= & \lambda x.a \mid N &
\end{array}
$$

Small step semantics for strict functional languages

$$\textit{Evaluation Contexts} \quad E \quad ::= \quad [\,] \mid E\,a \mid v\,E$$

$$\textsc{Beta}_v \qquad (\lambda x.a)\,v \to a[x/v]$$

$$\textsc{Context} \qquad \frac{a \to b}{E[a] \to E[b]}$$

Strategy and big-step semantics

Characteristics of the reduction strategy

Weak reduction: We cannot reduce under λ-abstractions;

Call-by-value: In an application $(\lambda x.a)\,b$, the argument b must be fully reduced to a value before β-reduction can take place.

Left-most reduction: In an application $a\,b$, we must reduce a to a value first before we can start reducing b.

Deterministic: For every term a, there is at most one b such that $a \to b$.

Big step semantics for strict functional languages

$$N \Rightarrow N \qquad\qquad \lambda x.a \Rightarrow \lambda x.a \qquad\qquad \frac{a \Rightarrow \lambda x.c \quad b \Rightarrow v_\circ \quad c[x/v_\circ] \Rightarrow v}{a\,b \Rightarrow v}$$

Interpreter

The big step semantics induces an efficient implementation

```
type term =
  Const of int | Var of string | Lam of string * term | App of term * term

exception Error

let rec subst x v = function        (* assumes v is closed *)
  | Const n -> Const n
  | Var y -> if x = y then v else Var y
  | Lam(y, b) -> if x = y then Lam(y, b) else Lam(y, subst x v b)
  | App(b, c) -> App(subst x v b, subst x v c)

let rec eval = function
  | Const n -> Const n
  | Var x -> raise Error
  | Lam(x, a) -> Lam(x, a)
  | App(a, b) ->
      match eval a with
      | Lam(x, c) -> let v = eval b in eval (subst x v c)
      | _ -> raise Error
```

Exercises

1. Define the small-step and big-step semantics for the call-by-name
2. Deduce from the latter the interpreter
3. Use the technique introduced for the type `'a delayed` earlier in the course to implement an interpreter with lazy evaluation.

Improving implementation

Environments

- Implementing textual substitution $a[x/v]$ is *inefficient*. This is why compilers and interpreters *do not* implement it.
- Alternative: record the binding $x \mapsto v$ in an *environment* e

$$\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow \lambda x.a$$

$$\frac{e \vdash a \Rightarrow \lambda x.c \quad e \vdash b \Rightarrow v_\circ \quad e; x \mapsto v_\circ \vdash c \Rightarrow v}{e \vdash a\,b \Rightarrow v}$$

Giving up substitutions in favor of environments does not come for free

- **Lexical scoping** requires careful handling of environments

```
let x = 1 in
let f = λy.(x+1) in
let x = "foo" in
f 2
```

In the environment used to evaluate `f 2` the variable `x` is bound to 1.

Exercise

Try to evaluate

```
let x = 1 in
let f = λy.(x+1) in
let x = "foo" in
f 2
```

by the big-step semantics in the previous slide,
where `let` x `=` a `in` b is syntactic sugar for $(\lambda x.b)a$

let us outline it together

Function closures

To implement *lexical scoping in the presence of environments*, function abstractions $\lambda x.a$ must not evaluate to themselves, but to a function *closure*: a pair $(\lambda x.a)[e]$ (ie, the function and the *environment of its definition*)

Big step semantics with environments and closures

$$\begin{array}{llll} \textit{Values} & v & ::= & N \mid (\lambda x.a)[e] \\ \textit{Environments} & e & ::= & x_1 \mapsto v_1 ; \ldots ; x_n \mapsto v_n \end{array}$$

$$\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow (\lambda x.a)[e]$$

$$\frac{e \vdash a \Rightarrow (\lambda x.c)[e_\circ] \quad e \vdash b \Rightarrow v_\circ \quad e_\circ ; x \mapsto v_\circ \vdash c \Rightarrow v}{e \vdash a\,b \Rightarrow v}$$

De Bruijn indexes

Identify variable not by names but by the number $\underline{n}$ of λ's that separate the variable from its binder in the syntax tree.

$$\lambda x.(\lambda y.y\,x)x \quad \text{is} \quad \lambda.(\lambda.\underline{0}\,\underline{1})\underline{0}$$

$\underline{n}$ is the variable bound by the n-th enclosing λ. Environments become sequences of values, the n-th value of the sequence being the value of variable $\underline{n-1}$.

$$\begin{array}{llcl} \textit{Terms} & a,b & ::= & N \mid \underline{n} \mid \lambda.a \mid a\,b \\ \textit{Values} & v & ::= & N \mid (\lambda.a)[e] \\ \textit{Environments} & e & ::= & v_0;v_1;\ldots;v_n \end{array}$$

$$\frac{e = v_0;\ldots;v_n;\ldots;v_m}{e \vdash \underline{n} \Rightarrow v_n} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda.a \Rightarrow (\lambda.a)[e]$$

$$\frac{e \vdash a \Rightarrow (\lambda.c)[e_\circ] \quad e \vdash b \Rightarrow v_\circ \quad v_\circ;e_\circ \vdash c \Rightarrow v}{e \vdash a\,b \Rightarrow v}$$

The canonical, efficient interpreter

```
# type term = Const of int | Var of int | Lam of term | App of term * term
   and value = Vint of int | Vclos of term * environment
   and environment = value list                           (* use Vec instead *)

# exception Error

# let rec eval e a =
    match a with
     | Const n -> Vint n
     | Var n -> List.nth e n                          (* will fail for open terms *)
     | Lam a -> Vclos(Lam a, e)
     | App(a, b) ->
         match eval e a with
         | Vclos(Lam c, e') ->
             let v = eval e b in
             eval (v :: e') c
         | _ -> raise Error



# eval [] (App ( Lam (Var 0), Const (2)));;         (*  (λx.x)2 → 2  *)
- : value = Vint 2
```

Note:To obtain improved performance one should implement environments by persistent extensible arrays: for instance by the `Vec` library by Luca de Alfaro.

Outline

Closure conversion

Goal: make explicit the construction of closures and the accesses to the environment part of closures.

Input: a fully-fledged functional programming language, with general functions (possibly having free variables) as first-class values.

Output: the same language where only closed functions (without free variables) are first-class values. Such closed functions can be represented at run-time as code pointers, just as in C for instance.

Idea: every function receives its own closure as an extra argument, from which it recovers values for its free variables. Such functions are closed. Function closures are explicitly represented as a tuple (closed function, values of free variables).

Uses: compilation; functional programming in Java (pre-8), ANSI C (nested functions are allowed in Gnu C), ...

Definition of closure conversion

$$
\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![\lambda x.a]\!] & = & \mathtt{tuple}(\lambda(c,x).\mathtt{let}\ \ x_1 = \mathtt{field}_1(c)\ \ \mathtt{in} \\
& & \qquad\qquad\qquad \vdots \\
& & \qquad\qquad\qquad \mathtt{let}\ \ x_n = \mathtt{field}_n(c)\ \ \mathtt{in} \\
& & \qquad\qquad\qquad [\![a]\!], \\
& & \qquad\quad x_1,\ldots,x_n) \\
& & \text{where } x_1, \ldots, x_n \text{ are the free variables of } \lambda x.a \\
[\![a\,b]\!] & = & \mathtt{let}\ \ c = [\![a]\!]\ \ \mathtt{in}\ \mathtt{field}_0(c)(c, [\![b]\!])
\end{array}
$$

The translation extends homomorphically to other constructs, e.g.

$$
\begin{array}{rcl}
[\![\mathtt{let}\ \ x = a\ \ \mathtt{in}\ \ b]\!] & = & \mathtt{let}\ \ x = [\![a]\!]\ \ \mathtt{in}\ \ [\![b]\!] \\
[\![a+b]\!] & = & [\![a]\!] + [\![b]\!]
\end{array}
$$

Example of closure conversion

Source program in Caml:

```
fun x lst ->
  let rec map f l =
     match l with
       [] -> []
     | hd :: tl -> f hd :: map f tl
  in
     map (fun y -> x + y) lst

- : int -> int list -> int list = <fun>
```

Result of partial closure conversion for the f argument of map:

```
fun x lst ->
  let rec map f lst =
     match lst with
       [] -> []
     | hd :: tl -> field0(f)(f,hd) :: map f tl
  in
     map tuple(λ(c,y). let x = field1(c) in x + y,
               x)
          lst
```

Closure conversion for recursive functions

In a recursive function $\mu f.\lambda x.a$, the body a needs access to f , that is, the closure for itself. This closure can be found in the extra function parameter that closure conversion introduces.

$$
\begin{array}{rcl}
[\![\mu f.\lambda x.a]\!] & = & \mathtt{tuple}(\lambda(f,x).\mathtt{let}\ x_1 = \mathtt{field}_1(f)\ \mathtt{in} \\
& & \qquad\qquad\qquad \vdots \\
& & \qquad\qquad\quad \mathtt{let}\ x_n = \mathtt{field}_n(f)\ \mathtt{in} \\
& & \qquad\qquad\quad [\![a]\!], \\
& & \qquad x_1, \ldots, x_n)
\end{array}
$$

where x_1 , . . . , x_n are the free variables of $\mu f.\lambda x.a$

Notice that f is free in a and thus in $[\![a]\!]$, but bound in $[\![\mu f.\lambda x.a]\!]$.

In other terms, regular functions $\lambda x.a$ are converted exactly like pseudo-recursive functions $\mu c.\lambda x.a$ where c is a variable not free in a.

Closure conversion in object-oriented style

If the target of the conversion is an object-oriented language in the style of Java, C#, we can use the following variant of closure conversion:

$$
\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![\lambda x.a]\!] & = & \texttt{new C}_{\lambda x.a}(x_1,...,x_n) \\
 & & \text{where } x_1,\ldots,x_n \text{ are the free variables of } \lambda x.a \\
[\![ab]\!] & = & [\![a]\!]\texttt{.apply}([\![b]\!])
\end{array}
$$

Closure conversion in object-oriented style

The class $C_{\lambda x.a}$ (one for each λ-abstraction in the source) is defined (in C#) as follows:

```
public class Cλx.a {
  protected internal Object x1, ..., xn;
  public Cλx.a(Object x1 , ... , Object xn ) {
    this.x1 = x1 ; ...; this.xn = xn ;
  }
  public Object apply(Object x) {
    return ⟦a⟧ ;
  }
}
```

Typing

In order to have a more precise typing the static types of the variables and of the function should be used instead of `Object`. In particular the method `apply` should be given the same input and return types as the encoded function.

Closures and objects

In more general terms:

- Closure $\approx$ Object with a single apply method
- Object $\approx$ Closure with multiple entry points

Both function application and method invocation compile down to self application:

$$\begin{array}{rcl} [\![\mathit{fun}\ \mathit{arg}]\!] & = & \mathtt{let}\ c = [\![\mathit{fun}]\!]\ \mathtt{in}\ \mathtt{field}_0(c)(c, [\![\mathit{arg}]\!]) \\ [\![\mathit{obj}.\mathit{meth}(\mathit{arg})]\!] & = & \mathtt{let}\ o = [\![\mathit{obj}]\!]\ \mathtt{in}\ o.\mathit{meth}(o, [\![\mathit{arg}]\!]) \end{array}$$

Where an object is interpreted as a record whose fields are methods which are parametrized by `self`.

First class closure

Modern OOL such as Scala and C# (added much later for Java in JDK 8) provide syntax to define closures, without the need to encode them.
For instance C# provides a `delegate` modifier to define closures:

```
public delegate int DComparer (Object x, Object y)
```

Defines a new distinguished type `DComparer` whose instances are functions from two objects to `int` (i.e., `DComparer` ≡ `(Object*Object)→int`)

Instances are created by passing to `new` a static or instance method (with compatible types):

```
DComparer mycomp = new DComparer(String.Comparer)
```

The closure `mycomp` can be passed around (wherever an argument of type `DComparer` is expected), or applied as in `mycomp("Scala","Java")`

First class closure

Actually in C# it is possible to define “lambda expressions”:
Here how to write $(\lambda(x,y).x+y)$ in C#:

```
(x,y) => x + y
```

Lambda expressions can be used to instantiate closures:

```
DComparer myComp = (x,y) => x + y
```

Delegates (roughly, function types) can be polymorphic:

```
public delegate TResult Func<TArg0, TResult>(TArg0 arg0)
```

The delegate can be instantiated as `Func<int,bool> myFunc` where `int` is an input parameter and `bool` is the return value. The return value is always specified in the last type parameter. `Func<int, string, bool>` defines a delegate with two input parameters, `int` and `string`, and a return type of `bool`.

```
Func<int, bool> myFunc = x => x == 5;
bool result = myFunc(4);            // returns false of course
```

Outline

Defunctionalization

Goal: like closure conversion, make explicit the construction of closures and the accesses to the environment part of closures. Unlike closure conversion, do not use closed functions as first-class values.

Input: a fully-fledged functional programming language, with general functions (possibly having free variables) as first-class values.

Output: any first-order language (no functions as values). Idea: represent each function value $\lambda x.a$ as a data structure `C(`$\mathtt{v}_1$`,...,`$\mathtt{v}_n$`)` where the constructor `C` uniquely identifies the function, and the constructor arguments $\mathtt{v}_1$,..., $\mathtt{v}_n$ are the values of the variables $x_1, \ldots, x_n$ free in the body of the function.

Uses: functional programming in Pascal, Ada, Basic, . . .

Definition of defunctionalization

$$\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![\lambda x.a]\!] & = & \mathtt{C}_{\lambda x.a}(x_1,\ldots,x_n) \\
& & \text{where } x_1,\ldots,x_n \text{ are the free variables of } \lambda x.a \\
[\![\mu f.\lambda x.a]\!] & = & \mathtt{C}_{\mu f.\lambda x.a}(x_1,\ldots,x_n) \\
& & \text{where } x_1,\ldots,x_n \text{ are the free variables of } \mu f.\lambda x.a \\
[\![ab]\!] & = & \mathtt{apply}([\![a]\!],[\![b]\!])
\end{array}$$

The difference between recursive and non-recursive functions is made in the definition of `apply`

(Other constructs: homomorphically.)

Definition of defunctionalization

The `apply` function collects the bodies of all functions and dispatches on its first argument. There is one case per function occurring in the source program.

```
let rec apply(fun ,arg ) =
   match fun with
    | C_{λx.a}(x1,...,xn ) -> let x = arg in ⟦ a ⟧
    | C_{μf.λy.b}(x1,...,xm) -> let f = fun in let y = arg in ⟦ b ⟧
    | ...
in ⟦program⟧
```

Note

Unlike closure conversion, this is a whole-program transformation.

Example

Defunctionalization of $(\lambda x.\lambda y.x)\,1\,2$:

```
let rec apply (fun, arg) =
   match fun with
   | C1()  -> let x = arg in C2(x)
   | C2(x) -> let y = arg in x
in
   apply(apply(C1(), 1), 2)
```

We write `C1` for $\mathtt{C}_{\lambda x.\lambda y.x}$ and `C2` for $\mathtt{C}_{\lambda y.x}$.

Outline

Syntax

Terms	a, b	$::=$	N	Numeric constant
		$\mid$	x	Variable
		$\mid$	$a\,b$	Application
		$\mid$	$\lambda x.a$	Abstraction
		$\mid$	$\texttt{raise}\ a$	Raise
		$\mid$	$\texttt{try}\ a\ \texttt{with}\ x \to b$	Try

Values	v	$::=$	$\lambda x.a \mid N$

Small step semantics for exceptions

$$\begin{array}{rcll}
(\texttt{try}\ \ v\ \ \texttt{with}\ \ x \to b) & \to & v & \\
(\texttt{try}\ \ \texttt{raise}\ \ v\ \ \texttt{with}\ \ x \to b) & \to & b[x/v] & \\
P[\texttt{raise}\ \ v] & \to & \texttt{raise}\ \ v & \quad \textit{if } P \neq [\,]
\end{array}$$

$$\frac{a \to b}{E[a] \to E[b]}$$

Exception propagation contexts P are like reduction contexts E but do not allow skipping past a `try ... with`

Reduction contexts:

$$E ::= [\,] \mid E\,a \mid v\,E \mid \texttt{raise}\ \ E \mid \texttt{try}\ \ E\ \ \texttt{with}\ \ x \to a \mid \ldots$$

Exception propagation contexts: (no `try_with`)

$$P ::= [\,] \mid P\,a \mid v\,P \mid \texttt{raise}\ \ P \mid \ldots$$

Reduction semantics for exceptions

Assume the current program is $p = E[\texttt{raise}\ \ v]$, that is, we are about to raise an exception. If there is a `try...with` that encloses the raise, the program will be decomposed as

$$p = E'[\texttt{try}\ \ P[\texttt{raise}\ \ v]\ \ \texttt{with}\ \ x \to b]$$

where P does not contain any `try...with` constructs (that encloses the hole). P$[\texttt{raise}\ \ v]$ head-reduces to raise v , and $E'[\texttt{try}\ \ [\,]\ \ \texttt{with}\ \ x \to b]$ is an evaluation context. The reduction sequence is therefore:

$$\begin{array}{rcl} p = E'[\texttt{try}\ \ P[\texttt{raise}\ \ v]\ \ \texttt{with}\ \ x \to b] & \to & E'[\texttt{try raise}\ \ v\ \ \texttt{with}\ \ x \to b] \\ & \to & E'[b[x/v]] \end{array}$$

If there are no `try` ... `with` around the raise, E is a *propagation context* and the reduction is therefore

$$p = E[\texttt{raise}\ \ v]\ \ \to\ \ \texttt{raise}\ \ v$$

Reduction semantics for exceptions

When considering reduction sequences, a fourth possible outcome of evaluation appears: termination on an uncaught exception.

- Termination: $a \rightarrow^{*} v$
- Uncaught exception: $a \rightarrow \texttt{raise}\ v$
- Divergence: $a \rightarrow^{*} a' \rightarrow \ldots$
- Error: $a \rightarrow a' \nrightarrow$ where $a \neq v$ and $a \neq \texttt{raise}\ v$.

Big step semantics for exception

In big step semantics, the evaluation relation becomes $a \Rightarrow r$ where evaluation *results* are $r ::= v \mid \texttt{raise}\ \ v$. Add the following rules for try...with:

$$\frac{a \Rightarrow v}{\texttt{try}\ \ a\ \ \texttt{with}\ \ x \rightarrow b \Rightarrow v} \qquad \frac{a \Rightarrow \texttt{raise}\ \ v \qquad b[x/v] \Rightarrow r}{\texttt{try}\ \ a\ \ \texttt{with}\ \ x \rightarrow b \Rightarrow r}$$

as well as exception propagation rules such as:

$$\frac{a \Rightarrow \texttt{raise}\ \ v}{ab \Rightarrow \texttt{raise}\ \ v} \qquad \frac{a \Rightarrow v' \qquad b \Rightarrow \texttt{raise}\ \ v}{ab \Rightarrow \texttt{raise}\ \ v}$$

Conversion to exception-returning style

Goal: get rid of exceptions.

Input: a functional language featuring exceptions (`raise` and `try...with`).

Output: a functional language with pattern-matching but no exceptions.

Idea: every expression a evaluates to either $\mathit{Val}(v)$ if a evaluates normally, or to $\mathit{Exn}(v)$ if a terminates early by raising exception v. $\mathit{Val}, \mathit{Exn}$ are datatype constructors.

Uses: giving semantics to exceptions; programming with exceptions in Haskell; reasoning about exceptions in theorem provers.

Definition of the transformation

$$\begin{array}{rcl}
[\![\texttt{raise}\ a]\!] & = & \texttt{match}\ [\![a]\!]\ \texttt{with} \\
 & & \mid \mathit{Exn}(x) \rightarrow \mathit{Exn}(x) \\
 & & \mid \mathit{Val}(x) \rightarrow \mathit{Exn}(x) \\
[\![\texttt{try}\ a\ \texttt{with}\ x \rightarrow b]\!] & = & \texttt{match}\ [\![a]\!]\ \texttt{with} \\
 & & \mid \mathit{Exn}(x) \rightarrow [\![b]\!] \\
 & & \mid \mathit{Val}(y) \rightarrow \mathit{Val}(y)
\end{array}$$

Definition of the transformation

$$
\begin{array}{rcl}
[\![N]\!] & = & \mathit{Val}(N) \\
[\![x]\!] & = & \mathit{Val}(x) \\
[\![\lambda x.a]\!] & = & \mathit{Val}(\lambda x.[\![a]\!]) \\
[\![\texttt{let } x = a \texttt{ in } b]\!] & = & \texttt{match } [\![a]\!] \texttt{ with } \mathit{Exn}(x) \to \mathit{Exn}(x) \mid \mathit{Val}(x) \to [\![b]\!] \\
[\![a\,b]\!] & = & \texttt{match } [\![a]\!] \texttt{ with} \\
& & \mid \mathit{Exn}(x) \to \mathit{Exn}(x) \\
& & \mid \mathit{Val}(x) \to \texttt{match } [\![b]\!] \texttt{ with} \\
& & \qquad\qquad \mid \mathit{Exn}(y) \to \mathit{Exn}(y) \\
& & \qquad\qquad \mid \mathit{Val}(y) \to x\,y
\end{array}
$$

Effect on types: if $a : \tau$ then $[\![a]\!] : [\![\tau]\!]$ where $[\![\tau_1 \to \tau_2]\!] = (\tau_1 \to [\![\tau_2]\!])$ `outcome` and $[\![\tau]\!] = \tau$ `outcome` otherwise and where `type 'a outcome = Val of 'a | Exn of exn`.

Example of conversion

Let *fun* and *arg* be two variables, then:

```
⟦ try fun arg with w -> 0 ⟧ =
     match
        match Val(fun) with
        | Exn(x) -> Exn(x)
        | Val(x) ->
            match Val(arg) with
            | Exn(y) -> Exn(y)
            | Val(y) -> x y
     with
     | Val(z) -> Val(z)
     | Exn(w) -> Val(0)
```

Notice that the two inner `match` can be simplified yielding

```
⟦ try fun arg with w -> 0 ⟧ =
     match fun arg with
     | Val(z) -> Val(z)
     | Exn(w) -> Val(0)
```

This transformation can be generalized by defining *administrative reductions*.

Administrative reductions

The naive conversion generates many useless match constructs over arguments whose shape $\mathit{Val}(\ldots)$ or $\mathit{Exn}(\ldots)$ is known at compile-time.

These can be eliminated by performing administrative reductions $\rightarrow$ at compile-time, just after the conversion:

Administrative reduction

$$\begin{array}{rcl}(\texttt{match}\;\; \mathit{Exn}(v)\;\; \texttt{with}\;\; \mathit{Exn}(x) \to b \mid \mathit{Val}(x) \to c) & \xrightarrow{\text{adm}} & b[x/v] \\ (\texttt{match}\;\; \mathit{Val}(v)\;\; \texttt{with}\;\; \mathit{Exn}(x) \to b \mid \mathit{Val}(x) \to c) & \xrightarrow{\text{adm}} & c[x/v]\end{array}$$

Correctness of the conversion

Define the conversion of a value $\mathcal{V}(v)$ as $\mathcal{V}(N) = N$ and $\mathcal{V}(\lambda x.a) = \lambda x.[\![a]\!]$.

Theorem

1. *If $a \Rightarrow v$, then $[\![a]\!] \Rightarrow \mathit{Val}(\mathcal{V}(v))$.*
2. *If $a \Rightarrow \texttt{raise}\;\; v$, then $[\![a]\!] \Rightarrow \mathit{Exn}(\mathcal{V}(v))$.*
3. *If $a \Uparrow$, then $[\![a]\!] \Uparrow$.*

Outline

State (imperative programming)

The word *state* in programming language theory refers to the distinguishing feature of imperative programming: the ability to assign (change the value of) variables after their definition, and to modify data structures in place after their construction.

References

A simple yet adequate way to model state is to introduce references: indirection cells / one-element arrays that can be modified in place.
The basic operations over references are:

ref a
: Create a new reference containing initially the value of a.

deref a also written $!a$
: Return the current contents of reference a.

assign a b also written $a := b$
: Replace the contents of reference a with the value of b.
Subsequent deref a operations will return this value.

Semantics of references

Semantics based on substitutions fail to account for sharing between references:

$$\texttt{let r = ref 1 in r := 2; !r} \not\to \texttt{(ref 1) := 2; !(ref 1)}$$

Left: the same reference r is shared between assignment and reading; result is 2.

Right: two distinct references are created, one is assigned, the other read; result is 1.

To account for sharing, we must use an additional level of indirection:

- *ref a* expressions evaluate to locations ℓ : a new kind of variable identifying references uniquely. (Locations ℓ are values.)
- A global environment called the store associates values to references.

Reduction semantics for references

The one-step reduction relation becomes $a \lhd s \to a' \lhd s'$
(read: in initial store s, a reduces to a' and updates the store to s')

$$\begin{array}{rcll}
(\lambda x.a)v \lhd s & \to & a[x/v] \lhd s & \\
\mathit{ref}\ v \lhd s & \to & \ell \lhd (s + \ell \mapsto v) & \text{where } \ell \notin \mathit{Dom}(s) \\
\mathit{deref}\ \ell \lhd s & \to & s(\ell) \lhd s & \\
\mathit{assign}\ \ell\ v \lhd s & \to & (\,) \lhd (s + \ell \mapsto v) &
\end{array}$$

$$\textsc{Context}\ \frac{a \lhd s \to a' \lhd s'}{E(a) \lhd s \to E(a') \lhd s'}$$

Notice that we also added a new value, $(\,)$, the result of a side-effect.

Exercise: define the evaluation contexts $E()$

Example of reduction sequence

Let us reduce the following term

$$\texttt{let}\ r = \texttt{ref}\ 3\ \texttt{in}\ r := !r + 1; !r$$

that is

$$\texttt{let}\ r = \texttt{ref}\ 3\ \texttt{in}\ \texttt{let}\ \_ = r := !r + 1\ \texttt{in}\ !r$$

(recall that `e1; e2` is syntactic sugar for `let _ = e1 in e2`)

In red: the active redex at every step.

$$\begin{aligned}
&\texttt{let}\ r = \texttt{ref}\ 3\ \texttt{in}\ \texttt{let}\ \_ = r := !r + 1\ \texttt{in}\ !r \lhd \varnothing\\
&\quad\to \texttt{let}\ r = \ell\ \texttt{in}\ \texttt{let}\ \_ = r := !r + 1\ \texttt{in}\ !r \lhd \ell \mapsto 3\\
&\quad\to \texttt{let}\ \_ = \ell := !\ell + 1\ \texttt{in}\ !\ell \lhd \ell \mapsto 3\\
&\quad\to \texttt{let}\ \_ = \ell := 3 + 1\ \texttt{in}\ !\ell \lhd \ell \mapsto 3\\
&\quad\to \texttt{let}\ \_ = \ell := 4\ \texttt{in}\ !\ell \lhd \ell \mapsto 3\\
&\quad\to \texttt{let}\ \_ = ()\ \texttt{in}\ !\ell \lhd \ell \mapsto 4\\
&\quad\to !\ell \lhd \ell \mapsto 4\\
&\quad\to 4 \lhd \ell \mapsto 4
\end{aligned}$$

Conversion to state-passing style

Goal: get rid of state.

Input: a functional language featuring references.

Output: a pure functional language.

Idea: every expression a becomes a function that
takes a run-time representation of the current store and
returns a pair (result value, updated store).

Uses: give semantics to references; program imperatively in Haskell; reason about imperative code in theorem provers.

Definition of the conversion

Core constructs

$$
\begin{array}{rcl}
[\![N]\!] & = & \lambda s.(N,s) \\
[\![x]\!] & = & \lambda s.(x,s) \\
[\![\lambda x.a]\!] & = & \lambda s.(\lambda x.[\![a]\!],s) \\
[\![\texttt{let}\ x=a\ \texttt{in}\ b]\!] & = & \lambda s.\texttt{match}\ [\![a]\!]s\ \texttt{with}\ (x,s')\to[\![b]\!]s' \\
[\![a\,b]\!] & = & \lambda s.\texttt{match}\ [\![a]\!]s\ \texttt{with}\ (x_a,s')\to \\
 & & \qquad \texttt{match}\ [\![b]\!]s'\ \texttt{with}\ (x_b,s'')\to x_a\,x_b\,s''
\end{array}
$$

Notice in particular that in the application we return $x_a\,x_b\,s''$ and not just $x_a\,x_b$. The reason is that λ-abstractions have their body translated. For instance $[\![(\lambda x.a)\,5]\!]$ reduces to $\lambda s.((\lambda x.[\![a]\!])\,5\,s)$. The application $(\lambda x.[\![a]\!])5$ thus returns the translation of a term, $[\![a]\!]$, that is a function that expects a state.

Definition of the conversion

Constructs specific to references

$$\begin{aligned}
[\![\texttt{ref}\ a]\!] &= \lambda s.\texttt{match}\ [\![a]\!]\, s\ \texttt{with}\ (x, s') \to \texttt{store\_alloc}\, x\, s' \\
[\![!a]\!] &= \lambda s.\texttt{match}\ [\![a]\!]\, s\ \texttt{with}\ (x, s') \to (\texttt{store\_read}\, x\, s', s') \\
[\![a := b]\!] &= \lambda s.\texttt{match}\ [\![a]\!]\, s\ \texttt{with}\ (x_a, s') \to \\
&\qquad \texttt{match}\ [\![b]\!]\, s'\ \texttt{with}\ (x_b, s'') \to (\varepsilon, \texttt{store\_write}\, x_a\, x_b\, s'')
\end{aligned}$$

The operations `store_alloc`, `store_read` and `store_write` provide a concrete implementation of the store. Any implementation of the data structure known as persistent extensible arrays will do.

Here ε represents the `()` value.

For instance we can use `Vec`, a library of extensible functional arrays by Luca de Alfaro. In that case we have that locations are natural numbers, a store is a vector `s` created by `Vec.empty`, a fresh location for the store s is returned by `Vec.length` s. Precisely, we have

$$\begin{aligned} \texttt{store\_alloc}\ v\ s &= (\,\texttt{Vec.length}\ s\,,\ \texttt{Vec.append}\ v\ s\,)\\ \texttt{store\_read}\ \ell\ s &= \texttt{Vec.get}\ \ell\ s\\ \texttt{store\_write}\ \ell\ v\ s &= \texttt{Vec.set}\ \ell\ v\ s \end{aligned}$$

Typing (assuming all values stored in references are of the same type `sval`):

$\texttt{store\_alloc} : \texttt{sval} \to \texttt{store} \to \texttt{location} \times \texttt{store}$

$\texttt{store\_read} : \texttt{location} \to \texttt{store} \to \texttt{sval}$

$\texttt{store\_write} : \texttt{location} \to \texttt{sval} \to \texttt{store} \to \texttt{store}$

where `location` is `int` and `store` is `Vec.t`.

Example of conversion

Administrative reductions: (where x, y, s, and s' are variables)

$$\begin{aligned}
(\texttt{match}\ (a,s)\ \texttt{with}\ (x,s')\to b) &\xrightarrow{\text{adm}} \texttt{let}\ x=a\ \texttt{in}\ b[s'/s]\\
(\lambda s.b)s' &\xrightarrow{\text{adm}} b[s/s']\\
\texttt{let}\ x=v\ \texttt{in}\ b &\xrightarrow{\text{adm}} b[x/v]\\
\texttt{let}\ x=y\ \texttt{in}\ b &\xrightarrow{\text{adm}} b[x/y]
\end{aligned}$$

(the first reduction replaces only the store since replacing also a for x may change de evaluation order: a must be evaluated before the evaluation of b)

Example of translation after administrative reductions:

Consider again the term

$$\texttt{let}\ r=\texttt{ref}\ 3\ \texttt{in}\ r:=!r+1;!r$$

We have

```
⟦ let r = ref 3 in let x = r := !r + 1 in !r ⟧  =
    λs. match  store_alloc 3 s with (r, s1) ->
      let t = store_ read r s1 in
      let u = t + 1 in
      match (ε , store_write r u s1) with (x, s2) -> (store_read r s2, s2)
```

Outline

Notion of continuation

Given a program p and a subexpression a of p, the *continuation* of a is the computation that remains to be done once a is evaluated to obtain the result of p.

It can be viewed as a function: (value of a) $\mapsto$ (value of p).

Example

Consider the program $p = (1+2)*(3+4)$.
The continuation of $a = (1+2)$ is $\lambda x.x*(3+4)$.
The continuation of $a' = (3+4)$ is $\lambda x.3*x$.
(Remember that 1 + 2 has already been evaluated to 3.)
The continuation of the whole program p is of course $\lambda x.x$

Continuations and reduction contexts

Continuations closely correspond with reduction contexts in small-step operational semantics:

Nota Bene

If $E[a]$ is a reduct of p, then the continuation of a is $\lambda x.E[x]$.

Example

Consider again $p = (1+2)*(3+4)$.

$$\begin{array}{rcl}
(1+2)*(3+4) & = & E_1[1+2] \text{ with } E_1 = [\,]*(3+4) \\
\to 3*(3+4) & = & E_2[3+4] \text{ with } E_2 = 3*[\,] \\
\to \quad 3*7 & = & E_3[3*7] \text{ with } E_3 = [\,] \\
\to \quad 21 & &
\end{array}$$

The continuation of $1+2$ is $\lambda x.E_1[x] = \lambda x.x*(3+4)$.
The continuation of $3+4$ is $\lambda x.E_2[x] = \lambda x.3*x$.
The continuation of $3*7$ is $\lambda x.E_3[x] = \lambda x.x$.

What continuations are for?

Historically continuations where introduced to define a denotational semantics for the `goto` statement in imperative programming

- Imagine we have a pure imperative programming language.
- As suggested by the state passing translation a program `p` of this language can be interpreted as a function that transforms states into states:

$$[\![\mathtt{p}]\!] : \mathcal{S} \to \mathcal{S}$$

- This works as long as we do not have `GOTO`.

- Consider the following spaghetti code in BASIC

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

- Idea: add to the interpretation of programs a further parameter: a continuation.
- In this framework a continuation is a function of type $\mathcal{S} \to \mathcal{S}$ since it takes the result of a statement (i.e. a state) and returns a new result (new state).

$$[\![\mathrm{p}]\!] : \mathcal{S} \to (\mathcal{S} \to \mathcal{S}) \to \mathcal{S}$$

- Every (interpretation of a) statement will do their usual modifications on the state they received and then will pass the resulting state to the continuations they received
- Only the `GOTO` behaves differently: it throws away the received continuation and use instead the continuation of the statement to go to.
- For instance the statement in line 50 will receive a state and a continuation and will pass the received state to the continuation of the instruction 20.

Continuations for compiler optimizations

Explicit continuations are inserted by some compiler for optimization:

```
(*       defines the product of all prime numbers <= n       *)

let rec prodprime n =                      (* bear with this       *)
    if n = 1                               (* horrible indentation *)
      then
         1
      else if
         isprime n                         (* receives k returns b *)
      then n * prodprime (n-1)             (* receives j returns p *)
      else prodprime (n-1);;               (* receives h returns q *)
```

The compiler adds (at function calls) points to control the flow of this function

1. `isprime` is given a return address `k` and returns a boolean `b` to it
2. The first `prodprime` call will return at point `j` an integer `p`
3. The second `prodprime` call will return at point `h` an integer `q`

Continuations for compiler optimizations

```
let rec prodprime(n,c) =
  if n = 1
  then
    c 1                       (* pass 1 to the current continuation c *)
  else
    let k b =                                (* continuation of isprime *)
      if b
      then
        let j p =                          (* continuation of prodprime *)
          let a = n * p in c a in
          let m = n - 1
        in prodprime(m,j)  (*call prodprime(n-1) with its continuation*)
      else
        let h q =                          (* continuation of prodprime *)
          c q in
          let i = n - 1
        in prodprime(i,h)  (*call prodprime(n-1) with its continuation*)
    in isprime(n,k)        (* call isprime(n) with its continuation k *)
```

Notice that we added variables m and i to store intermediate results

(this is called ANF, or A-normal form and was introduced by Sabry and Felleisen in '92, it simplifies CPS transformation since all function calls have either variables or constants as arguments)

Advantages

Explicit continuations bring several advantages:

- Tail recursion: `prodprime` is now tail recursive. Also the call that was already call recursive has trivial continuation (`h` is equivalent to `c`) that can be simplified:

```
let h q =
   c q in
   let i = n - 1              ⇒        let i = n - 1
in prodprime(i,h)                      in prodprime(i,c)
```

- Inlining: In languages that are strict and/or have side effects inlining is very difficult to do directly. Explicit continuations overcome all the problems since *all actual parameters to functions are either variables or constants* (never a non-trivial sub-expression)
- Dataflow analysis describes static propagation of values. Continuation make this flow explicit and easy this analysis (for detection of dead-code or register allocation).

Continuations as first-class values

The Scheme language offers a primitive `callcc` (call with current continuation) that enables a subexpression a of the program to capture its continuation (as a function 'value of a' $\mapsto$ 'value of the program') and manipulate this continuation as a first-class value.

The expression `callcc`$(\lambda k.a)$ evaluates as follows:

- The continuation of this expression is passed as argument to $\lambda k.a$.
- Evaluation of a proceeds; its value is the value of `callcc`$(\lambda k.a)$.
- If, during the evaluation of a *or later* (if we stored k somewhere or we passed it along), we evaluate `throw` k v, evaluation continues as if `callcc`$(\lambda k.a)$ returned v.
  That is, the continuation of the `callcc` expression is reinstalled and restarted with v as the result provided by this expression.

Using first-class continuations

Libraries for lists, sets, and other collection data types often provide an imperative iterator iter, e.g.

```
(* list_iter: ('a -> unit) -> 'a list -> unit *)

let rec list_iter f l =
  match l with
    | [] -> ()
    | head :: tail -> f head; list_iter f tail
```

Using first-class continuations

Using first-class continuations, an existing imperative iterator can be turned into a function that returns the first element of a collection satisfying a given predicate `pred` (of type `'a -> bool`).

```
let find pred lst =
    callcc (λk.
        list_iter
            (λx. if pred x then throw k (Some x) else ())
            lst;
        None)
```

If an element `x` is found such that `pred x` = `true`, then the throw causes `Some x` to be returned immediately as the result of `find pred lst`. If no such element exists, `list_iter` terminates normally, and `None` is returned.

Using first-class continuations

The previous example can also be implemented with exceptions. However, callcc adds the ability to *backtrack* the search.

```
let find pred lst =
    callcc (λk.
        list_iter
            (λx. if pred x
                 then callcc (λk'. throw k (Some(x, k')))
                 else ())
            lst;
        None)
```

When x is found such that pred x = true, the function find returns not only x but also a continuation k' which, when thrown, will cause backtracking: the search in lst restarts at the element following x. This is used as shown in the next function.

Using first-class continuations

The following use of `find` will print all list elements satisfying the predicate:

```
let printall pred lst =
    match find pred list with
    | None -> ()
    | Some(x, k) -> print_string x; throw k ()
```

The `throw k ()` restarts `find pred list` where it left the last time.

First-class continuations

`callcc` and other control operators are difficult to use directly ("the `goto` of functional languages"), but in combination with references, can implement a variety of interesting control structures:

- Exceptions (seen)
- Backtracking (seen)
- Generators for imperative iterators such as Python's and C# `yield` (next slides).
- Coroutines / cooperative multithreading (few slides ahead).
- Checkpoint/replay debugging (in order to save the intermediate state —*ie*, a checkpoint— of a process you can save the continuation).

Python's `yield`

`yield` inside a function makes the function a *generator* that when called returns an object of type *generator*. The object has a method `next` that executes the function till the expression `yield`, returns the value of the `yield`, and at the next call of `next`, starts again right after the `yield`.

```
>>> def gen_fibonacci():                    # Generator of Fibonacci suite
...     a, b = 1, 2
...     while True:
...         yield a
...         a, b = b, a + b
...
>>> fib = gen_fibonacci()
>>> for i in range(4):
...     print fib.next()
...
1
2
3
5
>>> fib.next()
8
>>> fib.next()
13
```

Python's `yield`

Actually the argument of a `for` loop is a generator object.
At each loop the for calls the `next` method of the generator. When the generator does not find a next yield and exits, then it raises a exception that makes the for exit.

```
>>> for i in fib:
...     print i
...
21
34
55
89
144
233
377
610
987
 ⋮
 ⋮
```

Simulate `yield` by `callcc`

```
let return = ref (Obj.magic None);;
let resume = ref (Obj.magic None);;
let fib () = callcc (fun kk -> return := kk;
   let a,b = ref 1, ref 2 in
      while true do
         callcc (fun cc -> resume := cc; throw !return !a);
         b := !a + !b;                          (* note:  a,b ← b,a+b *)
         a := !b - !a;
      done; 0
   )
val fib : unit -> int = <fun>
```

1. Use two references to store addresses to resume `fib` and return from it;
2. Save the return point in `return`
3. Save the resumption point in `resume`
4. Exit `fib()` by "going to" `return` and returning the value of `!a`
5. Adjust the types (the function must return an `int`)
6. Use `callcc(fun k -> return:=k; throw !resume ())` to resume

Example

```
# #load "callcc.cma";;
# open Callcc;;
# let return = ref (Obj.magic None);;
val return : '_a ref = contents = <poly>
# let resume = ref (Obj.magic None);;
val resume : '_a ref = contents = <poly>
# let fib() = callcc (fun kk -> return := kk;
    let a,b = ref 1, ref 2 in
        while true do
          callcc(fun cc -> (resume := cc; (throw !return !a)));
          b := !a + !b;
          a := !b - !a;
       done; 0)            ;;
val fib : unit -> int = <fun>
# fib();;
- : int = 1
# callcc (fun k -> return:=k; throw !resume ());;
- : int = 2
# callcc (fun k -> return:=k; throw !resume ());;
- : int = 3
# callcc (fun k -> return:=k; throw !resume ());;
- : int = 5
# callcc (fun k -> return:=k; throw !resume ());;
- : int = 8
# callcc (fun k -> return:=k; throw !resume ());;
- : int = 13
```

Exercise

Rewrite the previous program without the `Object.magic` so that the references contain values of type `'a Callcc.cont option` (verbose)

```
# #load "callcc.cma";;
# open Callcc;;
# let return = ref None;;
val return : '_a option ref = contents = None
# let resume = ref None;;
val resume : '_a option ref = contents = None

# let fib() = callcc (fun kk -> return := (Some kk);
  let a,b = ref 1, ref 2 in
  while true do
  callcc(fun cc -> (
           resume := (Some cc);
           let Some k = !return in (throw k !a)));
  b := !a + !b;
  a := !b - !a;
  done; 0);;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
None
val fib : unit -> int = <fun>

# fib();;
- : int = 1
# callcc (fun k -> return:= Some k; let Some k = !resume in throw k ());;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
None
- : int = 2
# callcc (fun k -> return:= Some k; let Some k = !resume in throw k ());;
```

Loop and tail-recursion can be encoded by `callcc`

```
let fib () = callcc (fun kk ->
    return := kk;
    let a,b = ref 1, ref 2 in
    callcc(fun cc -> resume := cc);
    b := !a + !b;
    a := !b - !a;
    throw !return !a)
```

So for instance we can avoid to call multiple times the throw ... just do not modify the return address

```
 # let x = fib () in
     if x < 100 then  (
        print_int x; print_newline();
        throw !resume ())
     else ();;
 1
 2
 3
 5
 8
 13
 21
```

Let us do it in a more functional way by using variables for `a` and `b`

```
# let resume = ref (Obj.magic None);;
val resume : '_a ref = contents = <poly>
# let fib () = callcc (fun kk ->
    let a,b = callcc(fun cc -> resume := cc ; (1,1) ) in
    throw kk (b,a+b) );;
val fib : unit -> int * int = <fun>
# let x,y = fib () in
     if x < 100 then  (
        print_int x; print_newline();
        throw !resume (x,y))
     else ();;
1
2
3
5
8
13
21
34
55
89
- : unit = ()
```

Exercise

Modify `fib()` so as it does not need the reference `resume` for the continuation.

Coroutines

Coroutines are more generic than subroutines.

Subroutines can return only once; coroutines can return (yield) several times. Next time the coroutine is called, the execution just after the yield call.

An example in pseudo-code

```
var q :=  new queue

coroutine produce
    loop
        while q is not full
           create some new items
           add the items to q
        yield to consume

coroutine consume
    loop
        while q is not empty
           remove some items from q
           use the items
        yield to produce
```

Implementing coroutines with continuations

```
coroutine process1 n =
   loop
      print "1: received "; print_ln n
      yield n+1 to process2
coroutine process2 n =
   loop
      print "2: received "; print_ln n
      yield n+1 to process1
in process1 0
```

In OCaml with `callcc`

```
callcc (fun init_k ->
  let curr_k = ref init_k in
  let communicate x =
    callcc (fun k ->
      let old_k = !curr_k in curr_k := k; throw old_k x) in
  let rec process1 n =
    print_string "1: received "; print_int n; print_newline();
    process1(communicate(n+1))
  and process2 n =
    print_string "2: received "; print_int n; print_newline();
    process2(communicate(n+1)) in
  process1(callcc(fun start1 ->
    process2(callcc(fun start2 ->
      curr_k := start2; throw start1 0)))))
```

Coroutines and generators

Generators are also a generalization of subroutines to define iterators

They look less expressive since the yield statement in a generator does not specify a coroutine to jump to: this is not the case:

```
generator produce
    loop
        while q is not full
           create some new items
           add the items to q
        yield consume

generator consume
    loop
        while q is not empty
           remove some items from q
           use the items
        yield produce

subroutine dispatcher
   var d := new dictionary ⟨ generator → iterator⟩
    d[produce] := start produce
    d[consume] := start consume
   var current := produce
   loop current := d[current].next()
```

Rationale

It is possible to implement coroutines on top of a generator facility, with the aid of a top-level dispatcher routine that passes control explicitly to child generators identified by tokens passed back from the generators

Generators are a much more commonly found language feature

A number of implementations of coroutines for languages with generator support but no native coroutines use this or a similar model: e.g. Perl 6, C#, Ruby, Python (prior to 2.5), ...

In OCaml there is Jérôme Vouillon's lightweight thread library (`Lwt`) that provides cooperative multi-threading. This can be implemented by coroutines (see the concurrency part of the course).

Reduction semantics for continuations

Keep the same reductions "$\rightarrow$" and the same context rules as before, and add the following rules for `callcc` and `throw`:

$$\begin{array}{rcl} E[\texttt{callcc}\ v] & \rightarrow & E[v(\lambda x.E[x])] \\ E[\texttt{throw}\ k\ v] & \rightarrow & kv \end{array}$$

(recall: the v argument of the `callcc` is a function that expects a continuation)

Same evaluation contexts E as before.

Example of reductions

$$
\begin{aligned}
&E[\texttt{callcc}(\lambda k.1 + \mathit{throw}\ k\ 0)] \\
&\quad \rightarrow E[(\lambda k.1 + \mathit{throw}\ k\ 0)(\lambda x.E[x])] \\
&\quad \rightarrow E[1 + \mathit{throw}\ (\lambda x.E[x])\ 0] \\
&\quad \rightarrow (\lambda x.E[x])0 \\
&\quad \rightarrow E[0]
\end{aligned}
$$

Note how `throw` discards the current context $E[1 + [\,]]$ and reinstalls the saved context E instead.

Outline

Conversion to continuation-passing style (CPS)

Goal: make explicit the handling of continuations.

Input: a call-by-value functional language with `callcc`.

Output: a call-by-value or call-by-name, pure functional language (no `callcc`).

Idea: every term a becomes a function $\lambda k \ldots$ that receives its continuation k as an argument, computes the value v of a, and finishes by applying k to v.

Uses: compilation of `callcc`; semantics; programming with continuations in Caml, Haskell, ...

CPS conversion: Core constructs

$$\begin{array}{rcl}
[\![N]\!] & = & \lambda k.kN \\
[\![x]\!] & = & \lambda k.kx \\
[\![\lambda x.a]\!] & = & \lambda k.k(\lambda x.[\![a]\!]) \\
[\![\texttt{let } x = a \texttt{ in } b]\!] & = & \lambda k.[\![a]\!](\lambda x.[\![b]\!]k) \\
[\![a\,b]\!] & = & \lambda k.[\![a]\!](\lambda x.[\![b]\!](\lambda y.x\,y\,k))
\end{array}$$

A function $\lambda x.a$ becomes a function of two arguments, x and the continuation k that will receive the value of a.

In $[\![a\,b]\!]$, the variable x (which must not be free in b) will be bound to the value returned by a and y to the value of b. As for the state passing conversion, $x\,y$ will return the *translation* of an expression, so a function that expects a continuation, which is why we apply its result to k

Effect on types:

if $a : \tau$ then $[\![a]\!] : ([\![\tau]\!] \to \texttt{answer}) \to \texttt{answer}$ where

$$\begin{array}{rcll}
[\![b]\!] & = & (b \to \texttt{answer}) \to \texttt{answer} & \text{for base types } b \\
[\![\tau_1 \to \tau_2]\!] & = & [\![\tau_1]\!] \to ([\![\tau_2]\!] \to \texttt{answer}) \to \texttt{answer} &
\end{array}$$

CPS conversion: Continuation operators

$$
\begin{aligned}
[\![\texttt{callcc}\ a]\!] &= \lambda k.[\![a]\!]\,k\,k \\
[\![\texttt{throw}\ a\ b]\!] &= \lambda k.[\![a]\!](\lambda x.[\![b]\!](\lambda y.x\,y))
\end{aligned}
$$

In `callcc` *a*, the function value returned by $[\![a]\!]$ receives the current continuation *k* both as its argument (first occurrence of *k*) and as its continuation (second occurrence of *k*).

In `throw` *a b*, we discard the current continuation *k* and apply directly the value of *a* (which is a continuation captured by `callcc`) to the value of *b* (the former being bound to *x* and the latter to *y*).

Administrative reductions

The CPS translation $[\![\ldots]\!]$ produces terms that are more verbose than those one would naturally write by hand. For instance, in the case of an application of a variable f to a variable x:

$$[\![f\ x]\!] = \lambda k.(\lambda k_1.k_1 f)(\lambda y_1.(\lambda k_2.k_2 x)(\lambda y_2.y_1 y_2 k))$$

instead of the more natural $\lambda k.f\,x\,k$. This clutter can be eliminated by performing β reductions at transformation time to eliminate the "administrative redexes" introduced by the translation. In particular, we have

$$(\lambda k.ku)(\lambda x.a) \xrightarrow{\text{adm}} (\lambda x.a)u \xrightarrow{\text{adm}} a[x/u]$$

whenever u is a value or variable.

Examples of CPS translation

$$[\![f(f\,x)]\!]$$
$$= \lambda k.fx(\lambda y.f\,y\,k)$$

$$[\![\mu\,fact.\lambda n.\texttt{if }\ n=0\ \texttt{ then }\ 1\ \texttt{ else }\ fact(n-1)*n]\!]$$
$$= \lambda k_0.k_0(\mu\,fact.\lambda n.\lambda k.\texttt{if }\ n=0\ \texttt{ then }\ k\,1\ \texttt{ else }\ fact(n-1)(\lambda v.k(v*n)))$$

Notice that the factorial function has become tail-recursive

Execution of CPS-converted programs

Execution of a program *prog* is achieved by applying its CPS conversion to the initial continuation $\lambda x.x$:

$$[\![prog]\!](\lambda x.x)$$

Theorem (Soundness)

If $a \to^* N$, then $[\![a]\!](\lambda x.x) \to^* N$.

CPS terms

The λ-terms produced by the CPS transformation have a very specific shape, described by the following grammar:

$$\begin{array}{lcll} \mathit{atom} & ::= & x \mid N \mid \lambda x.\mathit{body} \mid \lambda x.\lambda k.\mathit{body} & \textbf{CPS atom} \\ \mathit{body} & ::= & \mathit{atom} \mid \mathit{atom}_1\ \mathit{atom}_2 \mid \mathit{atom}_1\ \mathit{atom}_2\ \mathit{atom}_3 & \textbf{CPS body} \end{array}$$

$[\![a]\!]$ is an atom, and $[\![a]\!](\lambda x.x)$ is a body .

Reduction of CPS terms

$$\begin{array}{lcll} \mathit{atom} & ::= & x \mid N \mid \lambda v.\mathit{body} \mid \lambda x.\lambda k.\mathit{body} & \textbf{CPS atom} \\ \mathit{body} & ::= & \mathit{atom} \mid \mathit{atom}_1\ \mathit{atom}_2 \mid \mathit{atom}_1\ \mathit{atom}_2\ \mathit{atom}_3 & \textbf{CPS body} \end{array}$$

Note that all applications (unary or binary) are in tail-position and at application-time, their arguments are closed atoms, that is, values.

The following reduction rules suffice to evaluate CPS-converted programs:

$$\begin{array}{rcl} (\lambda x.\lambda k.\mathit{body})\mathit{atom}_1\ \mathit{atom}_2 & \to & \mathit{body}[x/\mathit{atom}_1, k/\mathit{atom}_2] \\ (\lambda x.\mathit{body})\mathit{atom} & \to & \mathit{body}[x/\mathit{atom}] \end{array}$$

These reductions are always applied at the top of the program—there is no need for reduction under a context.

CPS terms can be executed by a stackless abstract machine with three registers, an environment and a code pointer.

We will see it in detail in the part on Abstract Machines.
See also [*Compiling with continuations*, A. Appel, Cambridge University Press, 1992].

CPS conversion and reduction strategy

Theorem (Indifference (Plotkin 1975))

A closed CPS-converted program $[\![a]\!](\lambda x.x)$ evaluates in the same way in call-by-name, in left-to-right call-by-value, and in right-to-left call-by-value.

CPS conversion encodes the reduction strategy in the structure of the converted terms. The one we gave corresponds to left-to-right call-by-value.

$$[\![a\,b]\!] \quad = \quad \lambda k.[\![a]\!](\lambda x_a.[\![b]\!](\lambda x_b.x_a\,x_b\,k))$$

Right-to-left call-by-value is obtained by taking

$$[\![a\,b]\!] \quad = \quad \lambda k.[\![b]\!](\lambda x_b.[\![a]\!](\lambda x_a.x_a\,x_b\,k))$$

while call-by-name is achieved by taking

$$\begin{aligned} [\![x]\!] &= \lambda k.x\,k \\ [\![a\,b]\!] &= \lambda k.[\![a]\!](\lambda x_a.x_a[\![b]\!]k) \end{aligned}$$

Control operators and classical logic

Control operators such as `callcc` extend the Curry-Howard correspondence from *intuitionistic logic* to *classical logic*.

The *Pierce's law* $((P \to Q) \to P) \to P$ is not derivable in the intuitionistic logic while it is true in classical logic (in particular if we take $Q \equiv \bot$ then it becomes $((\neg P \to P) \to P$: if from $\neg P$ we can deduce P, then P must be true).

In terms of Curry-Howard it means that no term of the simply-typed λ-calculus has type $((P \to Q) \to P) \to P$.

But notice that

$$\texttt{callcc} : ((\alpha \to \beta) \to \alpha) \to \alpha$$

`callcc` takes as argument a function f of type $((\alpha \to \beta) \to \alpha)$ which can either return a value of type α directly or apply an argument of type α to the continuation of type $(\alpha \to \beta)$. Since the existing context is deleted when the continuation is applied, the type β (which is the type of the result of the whole program) is never used and may be taken to be $\bot$.

`callcc` is a proof for Pierce's law. It extends the Curry-Howard correspondence from intuitionistic logic to classical logic

It is therefore possible to "prove" the excluded middle axiom $\forall P.P \vee \neg P$.

Modulo Curry-Howard, this axiom corresponds to the type $\forall P.P + (P \to \mathit{False})$, where *False* is an empty type and $A + B$ is a datatype with two constructors
$\mathit{Left} : A \to A + B$ and $\mathit{Right} : B \to A + B$.
The following term "implements" (ie, it proves) excluded middle:

$$\texttt{callcc}(\lambda k.\mathit{Right}(\lambda p.\texttt{throw}\ k(\mathit{Left}(p))))$$

Exercise

Check that the term above proves the excluded middle

What about the CPS translation?

CPS and double negation

Let $\neg A = (A \to \bot)$ where $\bot$ represent "false". In intuitionistic logic

$$\vdash A \to \neg\neg A$$

whose proof is $\lambda x{:}A.\lambda f{:}\neg A.fx$. On the other hand:

$$\not\vdash \neg\neg A \to A$$

[this is the "reductio ad absurdum": if $\neg A$ implies $\bot$, then A; that is, $(\neg A \to \bot) \to A$]
It is not possible to define a closed λ-term of the type above.

However:

$$\vdash \neg\neg\neg A \to \neg A$$

whose proof is: $\lambda f : \neg\neg\neg A.\lambda x : A.f(\lambda g : \neg A.gx)$.
This suggests a *double negation* translation from classical to intuitionistic logic:

- $[\![\phi]\!] = \neg\neg\phi$ if ϕ is *atomic* (ie, a basic type)
- $[\![A \to B]\!] = [\![A]\!] \to [\![B]\!]$

CPS and double negation

Theorem (Glivenko 1929)

$$\vdash_{\mathit{classic}} A \quad \mathit{iff} \quad \vdash_{\mathit{intuitionistic}} [\![A]\!]$$

In terms of the Curry Howard isomorphism

$$\vdash_{\mathit{classic}} M : A \quad \text{iff} \quad \vdash_{\mathit{intuitionistic}} [\![M]\!] : [\![A]\!]$$

where $[\![M]\!]$ is (essentially) the CPS translation of M.

So the CPS translation extends the Curry-Howard isomorphism to the “double negation *encoding*” of the classical propositional logic

See A Formulæ-as-Types Notion of Control, T. Griffin, Symp. Principles of Programming Languages 1990.

References

- A. Appel. Programming with continuations.
- Slides of the course *Functional Programming Languages* by Xavier Leroy (from which the slides of this and the following part heavily borrowed) available on the web: `https://xavierleroy.org/mpri/2-4/transformations.2up.pdf`

Abstract machines

Outline

Execution models for a language

1. **Interpretation:** control (sequencing of computations) is expressed by a term of the source language, represented by a tree-shaped data structure. The interpreter traverses this tree during execution.
2. **Compilation to native code:** control is compiled to a sequence of machine instructions, before execution. These instructions are those of a real microprocessor and are executed in hardware.
3. **Compilation to abstract machine code:** control is compiled to a sequence of instructions. These instructions are those of an abstract machine. They do not correspond to that of an existing hardware processor, but are chosen close to the basic operations of the source language.Yet another example of program transformations between different languages

Next: short overview of abstract machines for functional languages

Outline

Abstract machine for arithmetic expressions

Arithmetic expressions:

$$a ::= N \mid a+a \mid a-a$$

Machine Components

1. A code pointer
2. A stack

Instruction set:

`CONST(`N`)`	push integer N on stack
`ADD`	pop two integers, push their sum
`SUB`	pop two integers, push their difference

Compilation scheme

Compilation (translation of expressions to sequences of instructions) is just translation to “reverse Polish notation”:

$$\begin{array}{rcl} [\![N]\!] & = & \mathtt{CONST}(N) \\ [\![a_1 + a_2]\!] & = & [\![\mathtt{a}_1]\!];[\![\mathtt{a}_2]\!];\mathtt{ADD} \\ [\![a_1 - a_2]\!] & = & [\![\mathtt{a}_1]\!];[\![\mathtt{a}_2]\!];\mathtt{SUB} \end{array}$$

Example

$[\![\ 5 - (1 + 2)]\!] = \mathtt{CONST(5)};\ \mathtt{CONST(1)};\ \mathtt{CONST(2)};\ \mathtt{ADD};\ \mathtt{SUB}$

Transitions

BEFORE		AFTER	
Code	Stack	Code	Stack
`CONST(`N`)` ; c	s	c	$N.s$
`ADD` ; c	$n_2.n_1.s$	c	$(n_1+n_2).s$
`SUB` ; c	$n_2.n_1.s$	c	$(n_1-n_2).s$

Let us try to execute the compilation of $5-(1+2)$ with an empty stack

Code	Stack
`CONST(5); CONST(1); CONST(2); ADD; SUB`	ε
`CONST(1); CONST(2); ADD; SUB`	5
`CONST(2); ADD; SUB`	1.5
`ADD; SUB`	2.1.5
`SUB`	3.5
ε	2

Notice the right-to-left execution order

Outline

SECD: abstract-machine for call by value

Machine Components

1. A code pointer
2. An environment
3. A stack

Instruction set: (+ previous arithmetic operations)

ACCESS(n)	push n-th field of the environment
CLOSURE(c)	push closure of code c with current environment
LET	pop value and add it to environment
ENDLET	discard first entry of environment
APPLY	pop function closure and argument, perform application
RETURN	terminate current function, jump back to caller

Historical note: (S)tack, (E)nvironment, (C)ontrol, (D)ump. (SCD) are implemented by stacks, (E) is an array. (C) is our code pointer, (D) is the return stack as in the first version of the ZAM later on.

Compilation scheme

$$\begin{aligned}
[\![\underline{n}]\!] &= \texttt{ACCESS}(n)\\
[\![\lambda a]\!] &= \texttt{CLOSURE}([\![a]\!]\ ;\ \texttt{RETURN})\\
[\![\texttt{let}\ a\ \texttt{in}\ b]\!] &= [\![a]\!]\ ;\ \texttt{LET}\ ;\ [\![b]\!]\ ;\ \texttt{ENDLET}\\
[\![ab]\!] &= [\![a]\!]\ ;\ [\![b]\!]\ ;\ \texttt{APPLY}
\end{aligned}$$

(constants and arithmetic as before)

Example

Term:

$(\lambda(\underline{0}+1))2$ (i.e., $(\lambda x.x+1)2$)

Code:

`CLOSURE(ACCESS(0);CONST(1);ADD;RETURN) ; CONST(2) ; APPLY`

Transitions

BEFORE			AFTER		
Code	Env	Stack	Code	Env	Stack
`ACCESS(n)`; c	e	s	c	e	$e(n).s$
`LET`; c	e	$v.s$	c	$v.e$	s
`ENDLET`; c	$v.e$	s	c	e	s
`CLOSURE(`c'`)`; c	e	s	c	e	$c'[e].s$
`APPLY`; c	e	$v.c'[e'].s$	c'	$v.e'$	$c.e.s$
`RETURN`; c	e	$v.c'.e'.s$	c'	e'	$v.s$

where $\mathrm{c}[\mathrm{e}]$ denotes the closure of code c with environment e.

Example

Code: `CLOSURE(`c`); CONST(2); APPLY`
where: c = `ACCESS(0);CONST(1);ADD;RETURN`

Code	Env	Stack
`CLOSURE(`c`); CONST(2); APPLY`	e	s
`CONST(2); APPLY`	e	$c[e].s$
`APPLY`	e	$2.c[e].s$
c	$2.e$	$\varepsilon.e.s$
`CONST(1);ADD;RETURN`	$2.e$	$2.\varepsilon.e.s$
`ADD;RETURN`	$2.e$	$1.2.\varepsilon.e.s$
`RETURN`	$2.e$	$3.\varepsilon.e.s$
ε	e	$3.s$

Soundness

Of course we always have to show that the compilation is correct, in the sense that it preserves the semantics of the reduction. This is stated as follows

Theorem (soundness of SECD)

If $e \Rightarrow v$ then the SECD machine in the state $([\![e]\!], \varepsilon, \varepsilon)$ reduces to the state $(\varepsilon, \varepsilon, \bar{v})$, where $\bar{v}$ is the machine value for v (the same integer for an integer, and the corresponding closure for a λ-abstraction.)

(where $\Rightarrow$ is the call-by-value, weak-reduction, big-step semantics defined in the “Refresher course on operational semantics”)

Outline

An optimization: tail call elimination

Consider:

```
    f = λ. ... g 1 ...
    g = λ. h(...)
    h = λ. ...
```

The call from g to h is a tail call: when h returns, g has nothing more to compute, it just returns immediately to f.

At the machine level, the code of g is of the form `...;APPLY;RETURN`
When g calls h, it pushes a return frame on the stack containing the code `RETURN`. When h returns (e.g. a value v_h), it jumps to this `RETURN` in g, which jumps to the continuation in f.

Tail-call elimination consists in avoiding this extra return frame and this extra `RETURN` instruction, enabling h to return directly to f, and saving stack space.

An optimization: tail call elimination

```
f = λ. ... g 1 ...
g = λ. h(...)
h = λ. ...
```

Code	Env	Stack
APPLY;RETURN$_g$	e	$v.c_h[e_h].c_f.e_f.s$
c_h	$v.e_h$	$(\mathtt{RETURN}_g).e.c_f.e_f.s$
$\vdots$	$\vdots$	$\vdots$
RETURN$_h$	e''	$v_h.(\mathtt{RETURN}_g).e.c_f.e_f.s$
RETURN$_g$	e	$v_h.c_f.e_f.s$
c_f	e_f	$v_h.s$

Tail-call elimination consists in avoiding this extra return frame and this extra RETURN instruction, enabling h to return directly to f, and saving stack space.

The importance of tail call elimination

Tail call elimination is important for recursive functions whose recursive calls are in tail position — the functional equivalent to loops in imperative languages:

```
let rec fact n accu =
     if n = 0 then accu else fact (n-1) (accu*n)
 in fact 42 1
```

With tail call elimination, this code runs in constant stack space.

Without tail call elimination, it consumes $O(n)$ stack space exactly as

```
let rec fact n = if n = 0 then 1 else n * fact (n-1)
 in fact 42
```

Hello stack overflows!

SECD with tail-call elimination

Machine Components: as before

Instruction set: as before plus

`TAILAPPLY` perform application without pushing the return frame

Compilation scheme:

Split the compilation scheme in two functions: $\mathcal{T}$ for expressions in tail call position, $\mathcal{C}$ for other expressions.

$$
\begin{array}{rcll}
\mathcal{T}[\![\texttt{let } a \texttt{ in } b]\!] & = & \mathcal{C}[\![a]\!];\texttt{LET};\mathcal{T}[\![b]\!] & \\
\mathcal{T}[\![ab]\!] & = & \mathcal{C}[\![a]\!];\mathcal{C}[\![b]\!];\texttt{TAILAPPLY} & \\
\mathcal{T}[\![a]\!] & = & \mathcal{C}[\![a]\!];\texttt{RETURN} & (\textit{otherwise}) \\
\mathcal{C}[\![n]\!] & = & \texttt{ACCESS}(n) & \\
\mathcal{C}[\![\lambda a]\!] & = & \texttt{CLOSURE}(\mathcal{T}[\![a]\!]) & \\
\mathcal{C}[\![\texttt{let } a \texttt{ in } b]\!] & = & \mathcal{C}[\![a]\!];\texttt{LET};\mathcal{C}[\![b]\!];\texttt{ENDLET} & \\
\mathcal{C}[\![ab]\!] & = & \mathcal{C}[\![a]\!];\mathcal{C}[\![b]\!];\texttt{APPLY} &
\end{array}
$$

Transitions

The TAILAPPLY instruction behaves like APPLY, but does not bother pushing a return frame to the current function

BEFORE			AFTER		
Code	Env	Stack	Code	Env	Stack
TAILAPPLY; c	e	$v.c'[e'].s$	c'	$v.e'$	s
APPLY; c	e	$v.c'[e'].s$	c'	$v.e'$	$c.e.s$

Note also that $\mathcal{T}[\![\texttt{let}\ a\ \texttt{in}\ b]\!]$ does not end by ENDLET, since every code produced by $\mathcal{T}[\![]\!]$ ends either by TAILAPPLY and RETURN, and both TAILAPPLY and RETURN throw the current environment away.

Back to the example

Code	Env	Stack
APPLY;RETURN$_g$	e	$v.c_h[e_h].c_f.e_f.s$
c_h	$v.e_h$	$(\mathtt{RETURN}_g).e.c_f.e_f.s$
$\vdots$	$\vdots$	$\vdots$
RETURN$_h$	e''	$v_h.(\mathtt{RETURN}_g).e.c_f.e_f.s$
RETURN$_g$	e	$v_h.c_f.e_f.s$
c_f	e_f	$v_h.s$

Code	Env	Stack
TAILAPPLY	e	$v.c_h[e_h].c_f.e_f.s$
c_h	$v.e_h$	$c_f.e_f.s$
$\vdots$	$\vdots$	$\vdots$
RETURN$_h$	e''	$v_h.c_f.e_f.s$
c_f	e_f	$v_h.s$

Outline

The Krivine Machine $\mathcal{K}$

Machine Components

1. A code pointer c
2. An environment e
3. A stack s

Difference: stacks and environments no longer contain values but "thunks". These are closures $c[e]$ for generic expressions (not just λ's) and represent "frozen" expressions that are to be evaluated.

Instruction set:

ACCESS(n)	start evaluating the thunk at the n-th position of the environment
PUSH(c)	push a thunk for code c
GRAB	pop one argument and cons it to the environment

Compilation scheme

- Application pushes the argument as a thunk (i.e., current expression + its environment) on the stack and evaluates the function.
- λ-abstraction grabs its argument(s) from the stack and evaluates its body

$$\begin{array}{rcl} [\![\underline{n}]\!] & = & \mathtt{ACCESS}(n) \\ [\![\lambda a]\!] & = & \mathtt{GRAB}\ ;\ [\![a]\!] \\ [\![ab]\!] & = & \mathtt{PUSH}([\![b]\!])\ ;\ [\![a]\!] \end{array}$$

Nota bene

- Close to lambda calculus: three instructions for three terms
- Implements call-by-name

$$((\lambda.a)[e])(b[e']) \to a[b[e'].e]$$

(λ-calculus with explicit substitutions)

Transitions

BEFORE			AFTER			
Code	Env	Stack	Code	Env	Stack	
$\texttt{ACCESS}(n); c$	e	s	$c'; c$	e'	s	if $e(n) = c'[e']$
$\texttt{GRAB}; c$	e	$c'[e'].s$	c	$c'[e'].e$	s	
$\texttt{PUSH}(c'); c$	e	s	c	e	$c'[e].s$	

In pure λ-calculus `ACCESS()` has no continuation c, so it is rather

BEFORE			AFTER			
Code	Env	Stack	Code	Env	Stack	
$\texttt{ACCESS}(n)$	e	s	c'	e'	s	if $e(n) = c'[e']$
⋮	⋮	⋮	⋮	⋮	⋮	

Soundness and efficiency

Soundness:

Krivine's machine is much closer to λ-calculus, so it has a stronger soundness result in the sense that every reduction step of the Krivine machine corresponds to a reduction step in the CBN λ-calculus (technically, to the CBN λ-calculus with explicit substitutions).
The soundness of SECD is stated just for the big-step semantics.

Efficiency:

Krivine's machine is highly inefficient

- Duplicated execution of the same expressions (call-by-value instead of call-by-need)
- Duplicated values stored on the heap (no mark compression)
- Redundant information for variables (it dumbly stores a variable with its closure, instead of storing directly the value the varible is bound to)
- Much more (see research papers).

Outline

Adding call-by-need

We add an indirection to a HEAP, which maps locations to closures.

Evironments map variables (De Brujin indexes) to locations of the heap.

A **value** ($\in \mathcal{V}al$) is a closure of the form `(GRAB;`c`)[`e`]` (compiles as before)

We split the rules for variables (`ACCESS()`) and lambda's (`GRAB;c`) in two:

BEFORE				AFTER				
Code	Env	Stack	Heap	Code	Env	Stack	Heap	
`ACCESS(`n`)`	e	s	h	c'	e'	s	h	(1)
`ACCESS(`n`)`	e	s	h	c'	e'	$\text{mrk}(\ell).s$	h	(2)
`GRAB;` c	e	$c'[e'].s$	h	c	$\ell.e$	s	$h\{\ell \mapsto c'[e']\}$	(3)
`GRAB;` c	e	$\text{mrk}(\ell).s$	h	`GRAB;`c	e	s	$h\{\ell \mapsto (\texttt{GRAB}; c)[e]\}$	(4)
`PUSH(`c'`);` c	e	s	h	c	e	$c'[e].s$	h	

(1) if $e(n) = \ell$ and $h(\ell) = c'[e'] \in \mathcal{V}al$ activate the value stored for n

(2) if $e(n) = \ell$ and $h(\ell) = c'[e'] \notin \mathcal{V}al$ activate expr and mark the stack

(3) ℓ is fresh grab the argument on the top of the stack and allocate on heap

(4) store in the heap the value computed for the location ℓ

Mark compression

In some situations in the stack may contain sequences of markers.

For example for $(\lambda z.(\lambda y.z(yz))z)(\lambda x.x)$ the machine reduces at some point to a stack of the form mrk(ℓ_2).mrk(ℓ_1) and both locations contain the closure $(\lambda x.x)[\,]$ (try it)

When a sequence of markers is popped from the stack, the same value is assigned to each heap location pointed to by the markers

Optimization: avoid creating sequences of markers by sharing the first marker and result location among closures that receive the same value.

Mark compression

Solution: Add one level of indirection

Before: Environments map variables to pointers to closures

Now: Environments map variables to pointers to pointers to closures

BEFORE				AFTER				
Code	Env	Stack	Heap	Code	Env	Stack	Heap	
$\mathtt{ACCESS}(n)$	e	s	h	c'	e'	s	h	(1)
$\mathtt{ACCESS}(n)$	e	$\mathrm{mrk}(\ell').s$	h	c'	e'	$\mathrm{mrk}(\ell').s$	$h\{e(n) \mapsto \ell'\}$	(2a)
$\mathtt{ACCESS}(n)$	e	s	h	c'	e'	$\mathrm{mrk}(\ell).s$	h	(2b)
$\mathtt{GRAB};c$	e	$c'[e'].s$	h	c	$\ell.e$	s	$h\{\ell \mapsto \ell' ; \ell' \mapsto c'[e']\}$	(3)
$\mathtt{GRAB};c$	e	$\mathrm{mrk}(\ell).s$	h	$\mathtt{GRAB};c$	e	s	$h\{\ell \mapsto (\mathtt{GRAB};c)[e]\}$	
$\mathtt{PUSH}(c');c$	e	s	h	c	e	$c'[e].s$	h	

(1) if $h(e(n))=\ell$ and $h(\ell)=c'[e'] \in \mathcal{V}al$

(2a) if $h(e(n))=\ell$ and $h(\ell)=c'[e'] \notin \mathcal{V}al$ map $e(n)$ to ℓ' and dealloc ℓ

(2b) if $h(e(n))=\ell$ and $h(\ell)=c'[e'] \notin \mathcal{V}al$ and $s \neq \mathrm{mrk}(\ell').s'$ proceed as before

(3) ℓ and ℓ' are fresh

Short circuiting for dereferencing

When the argument of a function is a variable deferencing is not efficient.

Consider $(\lambda x.Mx)N$.

1. We evaluate Mx in the environment $\{x \mapsto N[\,]\}$.
2. This pushes on the stack the closure $x[\{x \mapsto N[\,]\}]$: silly!
3. Much more efficient and clever to push directly on the stack the closure $N[\,]$ (i.e., the result of evaluating x in the environment $\{x \mapsto N[\,]\}$)

We short-circuit the deferencing of a variable in argument position.

An optimization already present in early implementations of Algol 60.

Rationale: now expressions in closures are never variables. They are
- either lambdas (the closure is a value)
- or applications (the closure is a "thunk", a frozen expression).

Short circuiting for dereferencing

1. We split the rule for application (`PUSH()`) in two cases: when the argument is a variable and when it is not
2. We modify the rule for lambdas, since heap allocation is now performed at the application (instead of `GRAB`)

BEFORE				AFTER				
Code	Env	Stack	Heap	Code	Env	Stack	Heap	
$\mathtt{ACC}(n)$	e	s	h	c'	e'	s	h	(1)
$\mathtt{ACC}(n)$	e	$\mathsf{mrk}(\ell').s$	h	c'	e'	$\mathsf{mrk}(\ell').s$	$h\{e(n)\mapsto\ell'\}$	(2a)
$\mathtt{ACC}(n)$	e	s	h	c'	e'	$\ell.s$	h	(2b)
$\mathtt{GRAB};c$	e	$\mathsf{arg}(\ell).s$	h	c	$\ell.e$	s	h	
$\mathtt{GRAB};c$	e	$\mathsf{mrk}(\ell).s$	h	$\mathtt{GRAB};c$	e	s	$h\{\ell\mapsto(\mathtt{GRAB};c)[e]\}$	
$\mathtt{PUSH}(\mathtt{ACC}(n));c$	e	s	h	c	e	$\mathsf{arg}(e(n)).s$	h	
$\mathtt{PUSH}(c');c$	e	s	h	c	e	$\mathsf{arg}(\ell').s$	$h\{\ell\mapsto\ell';\ell'\mapsto c'[e]\}$	(3)

wrote `ACC()` instead of `ACCESS()` for space reasons

(1) if $h(e(n))=\ell$ and $h(\ell)=c'[e']\in\mathcal{V}al$

(2a) if $h(e(n))=\ell$ and $h(\ell)=c'[e']\notin\mathcal{V}al$

(2b) if $h(e(n))=\ell$ and $h(\ell)=c'[e']\notin\mathcal{V}al$ and $s\not\equiv\mathsf{mrk}(\ell').s'$

(3) ℓ and ℓ' are fresh and $c'\not\equiv\mathtt{ACC}(n)$

Outline

Eval-apply vs. Push-enter

Real machines are more sophisticated (e.g., register allocation, garbage collection, ...) and there exist many more variants than the ones presented. See Marlow and Peyton Jones's JFP'06 paper for better approximation.

Peyton Jones classifies AM for functional languages based on two subtly different ways to evaluate a function application $f\ a\ b$:

- Push-enter: (e.g., Krivine)
  Push on stack the arguments a and b and *enter* the code of for f (that at some point will try to grab its arguments from the stack)
- Eval-apply: (e.g., SECD)
  Evaluate f (to a closure $c[e]$) and *apply* it to the right number of arguments (i.e., evaluate a and extend environment e with its result and, if f is binary, do the same with b)

The difference becomes significant for curried applications of functions whose arity is *not* statically known:

The problem with arity

```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith k [] [] = []
zipWith k (x:xs) (y:ys) = k x y : zipWith k xs ys
```

Here `k` can end up to be unary, binary, ternary ... or more:

1. $(\lambda x.x)(\lambda x.x)y$ (apply first to second)
2. $(\lambda x.\lambda y.x+y)xy$ (sum first and second)
3. $(\lambda x.\lambda y.\lambda z.z)xy$ (return a list of identities)

The arity of the function `k` is known only when it is bound to a closure.

Arity matching: match the function arity with the # of arguments available:

- Push-enter: the function, which statically knows its own arity, examines the stack to figure out how many arguments it has been passed, and where they are. the *callee* is responsible for arity matching
- Eval-apply: the caller, which statically knows what the arguments are, examines the function closure, extracts its arity, and makes an exact call to the function. the *caller* is responsible for arity matching

The problem with arity

Consider again `k x y`

- Push-enter:
  - if there are too few arguments, the function must construct a partial application and return.
  - if there are too many arguments, then only the required arguments are consumed, the rest of the arguments are left on the stack to be consumed later
- Eval-apply:
  - If k takes two arguments, call it straightforwardly.
  - If k takes one, call it passing x, and call the resulting function passing y;
  - if k takes more than two, build and return a closure for partial application k x y

Nota bene:

This holds only for calls of *unknown* functions. For known functions such as:

```
let g x y = x*y
 in g 3 4
```

any decent compiler must load the arguments 3 and 4 into registers, or on the stack, and call the code for g directly (no closures created) both in push/enter and eval/apply

Outline

Curried functions

In eval-apply the application of curried functions is costly

$$
\begin{aligned}
[\![f\,a_1 ... a_n]\!] &= [\![f]\!]\ ;\ [\![a_1]\!]\ ;\ \mathtt{APPLY}\ ;\ ...\ ;\ [\![a_n]\!]\ ;\ \mathtt{APPLY} \\
[\![\lambda^n.b]\!] &= \mathtt{CLOSURE(...(CLOSURE(}[\![b]\!]\mathtt{;RETURN)...);RETURN)}
\end{aligned}
$$

Before the body b of the function starts executing, the SECD:
- constructs $n-1$ intermediate, short-lived closures;
- performs $n-1$ calls that return immediately

In push-enter it is more efficient:

$$
\begin{aligned}
[\![f\,a_1 ... a_n]\!] &= \mathtt{PUSH(}[\![a_n]\!]\mathtt{)};...;\mathtt{PUSH(}[\![a_1]\!]\mathtt{)};[\![\mathtt{f}]\!] \\
[\![\lambda^n.b]\!] &= \underbrace{\mathtt{GRAB};...;\mathtt{GRAB}}_{n\ times};[\![\mathtt{b}]\!]
\end{aligned}
$$

Push all the arguments, enter the function that grabs the needed arguments and executes the body.

Let us try each technique on the application $(\lambda.\lambda.\lambda.\underline{0})2\,1\,0$

Curried function application in eval-apply

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] =$

`CLOSURE(CLOSURE(CLOSURE(ACCESS(0);RETURN);RETURN);RETURN);`
`CONST(2) ; APPLY ; CONST(1) ; APPLY ; CONST(0) ; APPLY`

Code	Env	Stack
`CLOSURE`(c_2);a_2	[]	ε
a_2	[]	c_2.[]
`APPLY`;a_1	[]	2.c_2.[]
c_2	2	a_1.[]
`RETURN`	2	c_1[2].a_1.[]
a_1	[]	c_1[2]
`APPLY`;a_0	[]	1.c_1[2]
c_1	1.2	a_0.[]
`RETURN`	1.2	c_0[1.2].a_0.[]
a_0	[]	c_0[1.2]
`APPLY`	[]	0.c_0[1.2]
c_0	0.1.2	ε.[]
`RETURN`	0.1.2	0.ε.[]
ε	[]	0

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] =$ `CLOSURE`(c_2)`;` a_2

where for $i = 1, 2$

c_0	$=$	`ACCESS(0) ; RETURN`
c_i	$=$	`CLOSURE`(c_{i-1}) `; RETURN`
a_0	$=$	`CONST(0) ; APPLY`
a_i	$=$	`CONST(i) ; APPLY ;` a_{i-1}

Curried function application in push-enter

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] = \texttt{PUSH(0);PUSH(1);PUSH(2);GRAB;GRAB;GRAB;ACCESS(0)}$

Code	Env	Stack
$\texttt{PUSH(0)};p_1$	[]	ε
$\texttt{PUSH(1)};p_0$	[]	0[]
$\texttt{PUSH(2)};g_3$	[]	1[].0[]
$\texttt{GRAB};g_2$	[]	2[].1[].0[]
$\texttt{GRAB};g_1$	2[]	1[].0[]
$\texttt{GRAB};g_0$	1[].2[]	0[]
$\texttt{ACCESS(0)}$	0[].1[].2[]	0[]
0	[]	ε

In short:

$[\![(\lambda.\lambda.\lambda.\underline{0})2\,1\,0]\!] = \texttt{PUSH(0)};p_1$

where for $i = 1,2,3$

$$\begin{array}{rcl} g_0 & = & \texttt{ACCESS(0)} \\ g_i & = & \texttt{GRAB}\ ;\ g_{i-1} \\ p_0 & = & \texttt{PUSH(2)}\ ;\ g_3 \\ p_1 & = & \texttt{PUSH(1)}\ ;\ p_0 \end{array}$$

Push-enter clearly wins

ZAM

Combine the call-by-value semantics with the push-enter model

The ZAM (Zinc Abstract Machine)

(The model underlying the bytecode interpretors of Caml Light and OCaml.)

A call-by-value, push-enter model where the caller pushes one or several arguments on the stack and the callee pops them and put them in its environment.

Needs special handling for

- partial applications: $(\lambda x.\lambda y.b)\ a$
- over-applications: $(\lambda x.x)\ (\lambda x.x)\ a$

The ZAM

Machine Components: as the SECD but where the stack is split into a argument stack and a return stack (as in Landin's original SECD)

Instruction set: as the SECD plus

GRAB	grab argument on the stack OR create a closure
PUSHMARK	push a mark to signal the last argument

minus LET, which is replaced by a GRAB.

Compilation scheme:

$$
\begin{array}{rcl}
\mathcal{C}[\![n]\!] & = & \texttt{ACCESS}(n) \\
\mathcal{C}[\![\lambda a]\!] & = & \texttt{CLOSURE}(\mathcal{T}[\![\lambda a]\!]) \\
\mathcal{C}[\![b\, a_1 \ldots a_n]\!] & = & \texttt{PUSHMARK};\mathcal{C}[\![a_n]\!];\ldots;\mathcal{C}[\![a_1]\!];\mathcal{C}[\![b]\!];\texttt{APPLY} \\
\mathcal{C}[\![\texttt{let}\ a\ \texttt{in}\ b]\!] & = & \mathcal{C}[\![a]\!];\texttt{GRAB};\mathcal{C}[\![b]\!];\texttt{ENDLET} \\
\mathcal{T}[\![n]\!] & = & \texttt{ACCESS}(n);\texttt{RETURN} \\
\mathcal{T}[\![\lambda a]\!] & = & \texttt{GRAB};\mathcal{T}[\![a]\!] \\
\mathcal{T}[\![b\, a_1 \ldots a_n]\!] & = & \texttt{PUSHMARK};\mathcal{C}[\![a_n]\!];\ldots;\mathcal{C}[\![a_1]\!];\mathcal{C}[\![b]\!];\texttt{TAILAPPLY} \\
\mathcal{T}[\![\texttt{let}\ a\ \texttt{in}\ b]\!] & = & \mathcal{C}[\![a]\!];\texttt{GRAB};\mathcal{T}[\![b]\!]
\end{array}
$$

Notice the left to right evaluation order for function application

Transitions

BEFORE				AFTER			
Code	Env	ArgStack	RetStack	Code	Env	ArgStack	RetStack
ACCESS(n);c	e	s	r	c	e	$e(n).s$	r
CLOSURE(c');c	e	s	r	c	e	$c'[e].s$	r
TAILAPPLY;c	e	$c'[e'].s$	r	c'	e'	s	r
APPLY;c	e	$c'[e'].s$	r	c'	e'	s	$c.e.r$
PUSHMARK;c	e	s	r	c	e	$\boxed{*}.s$	r
GRAB;c	e	$\boxed{*}.s$	$c'.e'.r$	c'	e'	$(\texttt{GRAB};c)[e].s$	r
GRAB;c	e	$v.s$	r	c	$v.e$	s	r
RETURN;c	e	$v.\boxed{*}.s$	$c'.e'.r$	c'	e'	$v.s$	r
RETURN;c	e	$c'[e'].s$	r	c'	e'	s	r
ENDLET;c	$v.e$	s	r	c	e	s	r

Nota Bene

1. Having a separate TAILAPPLY command no longer is strictly necessary since it has same behaviour as RETURN and could be replaced by it (we keep it to stress the places where only TAILAPPLY applies).
2. The code produced by $\mathcal{T}[\![a]\!]$ always ends either by RETURN or (equivalently) by TAILAPPLY

Krivine machine hidden in the ZAM

Call-by-name evaluation in the ZAM can be achieved with the following compilation scheme, isomorphic to that of Krivine's machine:

$$\begin{aligned}
\mathcal{N}[\![n]\!] &= \mathtt{ACCESS}(n);\mathtt{TAILAPPLY}\\
\mathcal{N}[\![\lambda a]\!] &= \mathtt{GRAB};\mathcal{N}[\![a]\!]\\
\mathcal{N}[\![b\,a]\!] &= \mathtt{CLOSURE}(\mathcal{N}[\![a]\!]);\mathcal{N}[\![b]\!]
\end{aligned}$$

The other ZAM instructions (and the mark $\boxed{*}$, and the return stack) are just extra call-by-value baggage.

Merging the two stacks

Return addresses can be put on the argument stack provided they are pushed before the arguments, along with the separation marks.
For that we compile using a continuation passing style:

$$\begin{array}{rcl}
\mathcal{C}[\![n]\!]k &=& \texttt{ACCESS}(n);k \\
\mathcal{C}[\![\lambda a]\!]k &=& \texttt{CLOSURE}(\mathcal{T}[\![\lambda a]\!]);k \\
\mathcal{C}[\![b\,a_1\ldots a_n]\!]k &=& \texttt{PUSHRETADDR}(k); \\
&& \mathcal{C}[\![a_n]\!](\ldots(\mathcal{C}[\![a_1]\!](\mathcal{C}[\![b]\!](\texttt{TAILAPPLY})))\ldots) \\
\mathcal{C}[\![\texttt{let}\ a\ \texttt{in}\ b]\!]k &=& \mathcal{C}[\![a]\!](\texttt{GRAB};\mathcal{C}[\![b]\!](\texttt{ENDLET};k)) \\
\\
\mathcal{T}[\![n]\!] &=& \texttt{ACCESS}(n);\texttt{RETURN} \\
\mathcal{T}[\![\lambda a]\!] &=& \texttt{GRAB};\mathcal{T}[\![a]\!] \\
\mathcal{T}[\![b\,a_1\ldots a_n]\!] &=& \mathcal{C}[\![a_n]\!](\ldots(\mathcal{C}[\![a_1]\!](\mathcal{C}[\![b]\!](\texttt{TAILAPPLY})))\ldots) \\
\mathcal{T}[\![\texttt{let}\ a\ \texttt{in}\ b]\!] &=& \mathcal{C}[\![a]\!](\texttt{GRAB};\mathcal{T}[\![b]\!])
\end{array}$$

(Facilitates exception handling, stack resizing, etc.)

Transitions

BEFORE			AFTER		
Code	Env	Stack	Code	Env	Stack
GRAB; c	e	$v.s$	c	$v.e$	s
GRAB; c	e	$\boxed{*}.c'.e'.s$	c'	e'	$(\texttt{GRAB}; c)[e].s$
RETURN; c	e	$v.\boxed{*}.c'.e'.s$	c'	e'	$v.s$
RETURN; c	e	$c'[e'].s$	c'	e'	s
PUSHRETADDR(c'); c	e	s	c	e	$\boxed{*}.c'.e.s$
TAILAPPLY; c	e	$c'[e'].s$	c'	e'	s
ACCESS(n); c	e	s	c	e	$e(n).s$
ENDLET; c	$v.e$	s	c	e	s
CLOSURE(c'); c	e	s	c	e	$c'[e].s$

Once more we can use RETURN instead of TAILAPPLY

Handling of curried applications

Consider the code in the closure for $\lambda.\lambda.\lambda.a$:

$$\texttt{GRAB; GRAB; GRAB; } \mathcal{T}[\![a]\!]$$

and recall that $\mathcal{T}[\![a]\!]$ finishes by a RETURN (or equivalently by a TAILAPPLY)

- **Total application to 3 arguments:**
  - The stack on entry is $v_1.v_2.v_3.\boxed{*}.c'.e'$
  - The three GRABs succeed yielding an environment $v_3.v_2.v_1.e$.
  - $\mathcal{T}[\![a]\!]$ is executed. It produces a value v and finishes by a RETURN
  - RETURN sees the stack $v.\boxed{*}.c'.e'$, reinstalls the caller $c'[e']$, and returns v to it
- **Partial application to 2 arguments:**
  - The stack on entry is $v_1.v_2.\boxed{*}.c'.e'$
  - The third GRAB fails and returns $(\texttt{GRAB};\mathcal{T}[\![a]\!])[v_2.v_1.e]$, representing the result of the partial application.
- **Over-application to 4 arguments:**

  The stack on entry is $v_1.v_2.v_3.v_4.\boxed{*}.c'.e'$
  - The three GRABs succeed yielding an environment $v_3.v_2.v_1.e$.
  - $\mathcal{T}[\![a]\!]$ is executed. It produces a value v and finishes by a RETURN
  - RETURN sees the stack $v.v_4.\boxed{*}.c'.e'$, and tail-applies v to v_4

  (v should be a closure or otherwise the over-application would be wrong and the machine stuck).

Outline

Stackeless Machine for CPS terms

The λ-terms produced by the CPS transformation have the following form:

$$\begin{array}{lcll} a & ::= & \underline{n} \mid N \mid \lambda.b \mid \lambda\lambda.b & \textbf{CPS atom} \\ b & ::= & a \mid a_1\, a_2 \mid a_1\, a_2\, a_3 & \textbf{CPS body} \end{array}$$

Machine Components:

A stackless abstract machine with:

- a code pointer c
- an environment e
- three registers R_1, R_2, R_3.

Instruction set:

$\texttt{ACCESS}_i\texttt{(n)}$	store n-th field of the environment in R_i
$\texttt{CONST}_i\texttt{(}N\texttt{)}$	store the integer N in R_i
$\texttt{CLOSURE}_i\texttt{(c)}$	store closure of c in R_i
TAILAPPLY1	apply closure in R_1 to argument R_2
TAILAPPLY2	apply closure in R_1 to arguments R_2, R_3

Compilation scheme

Compilation of atoms $\mathcal{A}_i[\![a]\!]$ (leaves the value of a in R_i):

$$\begin{array}{rcl}
\mathcal{A}_i[\![\underline{n}]\!] & = & \mathtt{ACCESS}_i\mathtt{(}n\mathtt{)} \\
\mathcal{A}_i[\![N]\!] & = & \mathtt{CONST}_i\mathtt{(}N\mathtt{)} \\
\mathcal{A}_i[\![\lambda.b]\!] & = & \mathtt{CLOSURE}_i\mathtt{(}\mathcal{B}[\![b]\!]\mathtt{)} \\
\mathcal{A}_i[\![\lambda\lambda.b]\!] & = & \mathtt{CLOSURE}_i\mathtt{(}\mathcal{B}[\![b]\!]\mathtt{)}
\end{array}$$

Compilation of bodies $\mathcal{B}[\![b]\!]$:

$$\begin{array}{rcl}
\mathcal{B}[\![a]\!] & = & \mathcal{A}_1[\![a]\!] \\
\mathcal{B}[\![a_1\,a_2]\!] & = & \mathcal{A}_1[\![a_1]\!];\mathcal{A}_2[\![a_2]\!];\mathtt{TAILAPPLY1} \\
\mathcal{B}[\![a_1\,a_2\,a_3]\!] & = & \mathcal{A}_1[\![a_1]\!];\mathcal{A}_2[\![a_2]\!];\mathcal{A}_3[\![a_3]\!];\mathtt{TAILAPPLY2}
\end{array}$$

Transitions

BEFORE					AFTER				
Code	Env	R_1	R_2	R_3	Code	Env	R_1	R_2	R_3
$\texttt{TAILAPPLY1}; c$	e	$c'[e']$	v	-	c'	$v.e'$	-	-	-
$\texttt{TAILAPPLY2}; c$	e	$c'[e']$	v_1	v_2	c'	$v_2.v_1.e'$	-	-	-
$\texttt{ACCESS}_1(n); c$	e	—	v_2	v_3	c	e	$e(n)$	v_2	v_3
$\texttt{CONST}_1(N); c$	e	—	v_2	v_3	c	e	N	v_2	v_3
$\texttt{CLOSURE}_1(c'); c$	e	—	v_2	v_3	c	e	$c'[e]$	v_2	v_3
$\texttt{ACCESS}_2(n); c$	e	—	v_2	v_3	c	e	v_1	$e(n)$	v_3
$\texttt{CONST}_2(N); c$	e	—	v_2	v_3	c	e	v_1	N	v_3
$\texttt{CLOSURE}_2(c'); c$	e	—	v_2	v_3	c	e	v_1	$c'[e]$	v_3
$\texttt{ACCESS}_3(n); c$	e	—	v_2	v_3	c	e	v_1	v_2	$e(n)$
$\texttt{CONST}_3(N); c$	e	—	v_2	v_3	c	e	v_1	v_2	N
$\texttt{CLOSURE}_3(c'); c$	e	—	v_2	v_3	c	e	v_1	v_2	$c'[e]$

Continuations vs. stacks

That CPS terms can be executed without a stack is not surprising, given that the stack of a machine such as the SECD is isomorphic to the current continuation in a CPS-based approach.

```
f x = 1 + g x      g x = 2 - h x      h x = ...
```

Consider the execution point where h is entered. In the CPS model, the continuation at this point is

$$k = \lambda v.k'(2 - v) \text{ with } k' = \lambda v.k''(1 + v) \text{ and } k'' = \lambda v.v$$

In the SECD model, the stack at this point is

$$\underbrace{(\mathtt{SUB}\ ;\ \mathtt{RETURN}).e_g.2}_{\simeq k}\,.\,\underbrace{(\mathtt{ADD}\ ;\ \mathtt{RETURN}).e_f.1}_{\simeq k'}\,.\,\underbrace{\varepsilon}_{\simeq k''}$$

Continuations vs. stacks

At the machine level, stacks and continuations are two ways to represent the call chain: the chain of function calls currently active.

- Continuations: as a singly-linked list of heap-allocated closures, each closure representing a function activation (in the example of the previous slide $k \mapsto \lambda v.k'(2-v)[k' \mapsto \lambda v.k''(1+v)[k'' \mapsto \lambda v.v]]$).
  These closures are reclaimed by the garbage collector.
- Stacks: as contiguous blocks in a memory area outside the heap, each block representing a function activation. These blocks are explicitly deallocated by RETURN instructions.

Stacks are more efficient in terms of GC costs and memory locality, but need to be copied in full to implement `callcc`.

References

- Jean-Louis Krivine: A call-by-name lambda-calculus machine. Higher-Order and Symbolic Computation 20(3):199-207 (2007)
- Improving the lazy Krivine machine, by D. Friedman, A. Ghuloum, J. Siek, O. Winebarger. Higher-Order Symb Comput (2007)20:271-293.
- Making a fast curry: push/enter vs. eval/apply for higher-order languages by Simon Marlow and Simon Peyton Jones. ICFP '04 and JFP '06
- A. Appel. Compiling with continuations (Chapter 13).
- Simon Peyton Jones. The Spineless Tagless G machine. 1992 (the original STG virtual machine for Haskell, now outdated).
- Slides of the course *Functional Programming Languages* by Xavier Leroy (from which the slides of this and the following part **heavily** borrowed) available on the web:
  `https://xavierleroy.org/mpri/2-4/machines.2up.pdf`

Monads

Outline

Monads

Exception-returning style, state-passing style, and continuation-passing style of the previous part are all special cases of *monads*

Monads are thus a technical device that factor out commonalities between many program transformations ...

... but this is just one possible viewpoint. Besides that, they can be used

- To structure denotational semantics and make them easy to extend with new language features. (E. Moggi, 1989.)
- As a powerful programming techniques in pure functional languages, primary in Haskell. (P. Wadler, 1992).

Outline

Invent your first monad

Probably the best way to understand monads is to define one. Or better, arrive to a point where you realize that you need one (even if you do not know that it is a monad).

Many of the problems that monads try to solve are related to the issue of side effects. So we'll start with them.

Side Effects: Debugging Pure Functions

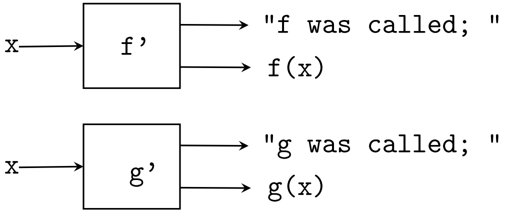
Input: We have functions `f` and `g` that both map floats to floats.

```
f,g : float -> float
```

Goal: Modify these functions to output their calls for debugging purposes

If we do not admit side effects, then the modified version `f'` and `g'` must return the output

```
f',g' : float -> float * string
```



We can think of these as 'debuggable' functions.

Binding

Problem: How to debug the composition of two 'debuggable' functions?

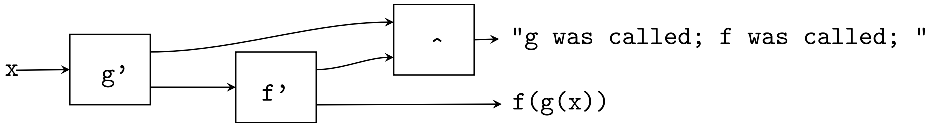
Intuition: We want the composition to have type `float -> float * string` but types no longer work!

Solution: Use concatenation for the debug messages and add some plumbing

```
let (y,s) = g' x in
let (z,t) = f' y in (z,s^t)
```

(*where ^ denotes string concatenation*)

Diagrammatically:



The bind function

Plumbing is ok ... once. To do it uniformly we need a higher-order function doing the plumbing for us. We need a function bind that upgrades f' so that it can be plugged in the output of g'. That is, we would like:

```
bind f' : (float*string) -> (float*string)
```

which implies that

```
bind : (float ->  (float*string)) -> ( (float*string) -> (float*string))
```

bind must

1. apply f' to the correct part of g' x and
2. concatenate the string returned by g' with the string returned by f'.

Exercise

Write the function bind.

```
# let bind f' (gx,gs) = let (fx,fs) = f' gx in (fx,gs^fs)
val bind : ('a -> 'b * string) -> 'a * string -> 'b * string = <fun>
```

The `return` function

Given two debuggable functions, `f'` and `g'`, now they can be composed by `bind`

`(bind f') . g'` (where "`.`" is Haskell's infix composition).

Write this composition as `f'`$\circ$ `g'`.

We look for a "debuggable" identity function `return` such that for every debuggable function `f` one has `return` $\circ$ `f = f` $\circ$ `return = f`.

Exercise

Define return.

```
# let return x = (x,"");;
val return : 'a -> 'a * string = <fun>
```

In Haskell (from now on we switch to this language):

```
Prelude> let return x = (x,"")
Prelude> :type return
return :: t -> (t, [Char])  --t is a schema variable, String = Char list
```

In summary, the function `return` lifts the result of a function into the result of a "debuggable" function.

The `lift` function

The return allows us to “lift” any *function* into a debuggable one:

`let lift f = return . f` (of type `(a -> b) -> a -> (b, [Char])`)

that is (in Ocaml) `let lift f x = (f x,"")`

The lifted version does much the same as the original function and, quite reasonably, it produces the empty string as a side effect.

Exercise

Show that `lift f` $\circ$ `lift g = lift (f.g)`

Summary

The functions, `bind` and `return`, allow us to compose debuggable functions in a straightforward way, and compose ordinary functions with debuggable functions in a natural way.

We just defined our first monad
Let us see more examples

Outline

A Container: Multivalued Functions

Consider `sqrt` and `cbrt` that compute the square root and cube root of a real number:

```
 sqrt,cbrt :: Float -> Float
```

Consider the complex version for these functions. They must return *lists* of results (two square roots and three cube roots)[1]

```
sqrt',cbrt' :: Complex  -> [Complex]
```

since they are *multi-valued* functions.

We can compose `sqrt` and `cbrt` to obtain the sixth root function

```
sixthrt x = sqrt (cbrt x)
```

Problem How to compose `sqrt'` and `cbrt'`?

Bind

We need a `bind` function that lifts `cbrt'` so that it can be applied to *all* the results of `sqrt'`

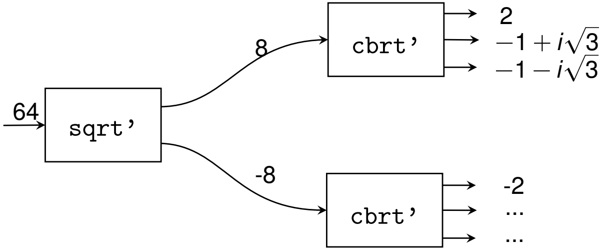
[1] `Complex` should be instead written `Complex Float`, since it is a Haskell module

bind for multivalued functions

Goal:

```
bind :: (Complex  -> [Complex]) -> ([Complex] -> [Complex])
```

Diagrammatically:



Exercise

Write an implementation of bind

Solution:

```
bind f x = concat (map f x)
```

`return` for multivalued functions

Again we look for an identity function for multivalued functions: it takes a result of a normal function and transforms it into a result of multi-valued functions:

```
  return :: a -> [a]
```

Exercise

Define `return`

Solution:

```
return x = [x]
```

Again

$$\mathtt{f} \circ \mathtt{return} = \mathtt{return} \circ \mathtt{f} = \mathtt{f}$$

while `lift f = return . f` transforms an ordinary function into a multivalued one: `lift :: (a -> b) -> a -> [b]`

We just defined our second monad
Let us see a last one and then recap

A more complex side effect: Random Numbers

The Haskell random function looks like this

```
random :: StdGen → (a,StdGen)
```

- To generate a random number you need a seed (of type `StdGen`)
- After you've generated the number you update the seed to a new value
- In a non-pure language the seed can be a global reference. In Haskell the new seed needs to be passed in and out explicitly.

So a function of type `a -> b` that needs random numbers must be lifted to a "randomized" function of type `a -> StdGen -> (b,StdGen)`

Exercise

1. Write the type of the `bind` function to compose two "randomized" functions.
2. Write an implementation of `bind`

A more complex side effect: Random Numbers

Solution:

1. `bind :: (a→StdGen→(b,StdGen))`
   `→(StdGen→(a,StdGen))→(StdGen → (b,StdGen))`
2. `bind f x seed = let (x',seed') = x seed in f x' seed'`

Exercise

Define the 'identity' randomized function. This needs to be of type

```
return :: a → (StdGen → (a,StdGen))
```

and should leave the seed unmodified.

Solution

```
return x g = (x,g)
```

Again, `lift f = return . f` turns an ordinary function into a randomized one that leaves the seed unchanged.

While $\mathtt{f} \circ \mathtt{return} = \mathtt{return} \circ \mathtt{f} = \mathtt{f}$ and $\mathtt{lift}\,\mathtt{f} \circ \mathtt{lift}\,\mathtt{g} = \mathtt{lift}(\mathtt{f}.\mathtt{g})$ where $\mathtt{f} \circ \mathtt{g} = (\mathtt{bind}\,\mathtt{f}).\mathtt{g}$

Outline

Monads

Step 1: Transform a type a into the type of particular *computations* on a.

```
-- The debuggable computations on a
type Debuggable a = (a,String)
-- The multivalued computation on a
type Multivalued a = [a]
-- The randomized computations on a
type Randomized a = StdGen -> (a,StdGen)
```

Step 2: Define the "plumbing" to lift functions on given types into functions on the "m computations" on these types where "m" is either Debuggable, or Multivalued, or Randomized.

```
bind :: (a -> m b) -> (m a -> m b)
return :: a -> m a
```

with $\mathtt{f} \circ \mathtt{return} = \mathtt{return} \circ \mathtt{f} = \mathtt{f}$ and $\mathtt{lift\ f} \circ \mathtt{lift\ g} = \mathtt{lift\ (f.g)}$, where '$\circ$' and lift are defined in terms of return and bind.

Monad

A *monad* is a triple formed by a type constructor m and two functions bind and return whose type and behavior is as described above.

Monads in Haskell

In Haskell, the `bind` function:

- it is written >>=
- it is infix
- its type is `m a -> (a -> m b) -> m b` (arguments are inverted)

This can be expressed by typeclasses:

```
class Monad m where
   -- chain computations
 (>>=)  :: m a -> ( a -> m b) -> m b
   -- inject
 return :: a -> m a
```

The properties of bind and return cannot be enforced, but monadic computation demands that the following equations hold

$$\begin{aligned}
\texttt{return}\ x \ \texttt{>>=}\ f &\equiv f\,x \\
m \ \texttt{>>=}\ \mathit{return} &\equiv m \\
m \ \texttt{>>=}\ (\lambda x.(f\,x \ \texttt{>>=}\ g)) &\equiv (m \ \texttt{>>=}\ f) \ \texttt{>>=}\ g
\end{aligned}$$

Monad laws

We already saw some of these properties:

$$\texttt{return}\ x\ \texttt{>>=}\ f \quad\equiv\quad f\,x \tag{1}$$

$$m\ \texttt{>>=}\ \mathit{return} \quad\equiv\quad m \tag{2}$$

$$m\ \texttt{>>=}\ (\lambda x.f\,x\ \texttt{>>=}\ g) \quad\equiv\quad (m\ \texttt{>>=}\ f)\ \texttt{>>=}\ g \tag{3}$$

Let us rewrite them in terms of our old `bind` function (with the different argument order we used before)

1. In (1) abstract the x then you have the *left identity*:

$$(\texttt{bind}\ f).\texttt{return} = f \circ \texttt{return} = f$$

2. In (2) consider $m = gx$ and abstract the x then you have the *right identity*

$$(\texttt{bind}\ \texttt{return}).g = \texttt{return} \circ g = g$$

3. Law (3) express *associativity* (exercise: prove it)

$$h \circ (f \circ g) = (h \circ f) \circ g$$

Writer, List and State Monads

The monads we showed are special cases of Writer, List, and State monads. Let us see their (simplified) versions

```
-- The Writer Monad
data Writer a = Writer (a, [Char])

instance Monad Writer where
   return x               = Writer (x,[])
   Writer (x,l) >>= f     = let Writer (x',l') = f x in Writer (x', l++l')


-- The List monad ([] data type is predefined)
instance Monad [] where
    return x              = [x]
    m >>= f               = concat (map f m)


-- The State Monad
data State s a = State (s -> (a,s))

instance Monad (State s) where
    return a              = State (λs -> (a,s))                -- \s -> (a,s)
    (State g) >>= f       = State (λs -> let (v,s') = g s in
                                         let State h = f v in h s')
```

Back to program transformations

QUESTION

Haven't you already seen the state monad?

Let us strip out the type constructor part:

```
return a     = λs -> (a,s)
a >>= f      = λs -> let (v,s') = a s in (f v) s'
```

It recalls somehow the transformation for the state passing style:

$$
\begin{array}{rcl}
[\![N]\!] & = & \lambda s.(N,s) \\
[\![x]\!] & = & \lambda s.(x,s) \\
[\![\lambda x.a]\!] & = & \lambda s.(\lambda x.[\![a]\!],s) \\
[\![\texttt{let}\ \ x = a\ \ \texttt{in}\ \ b]\!] & = & \lambda s.\texttt{match}\ \ [\![a]\!]s\ \ \texttt{with}\ \ (x,s') \to [\![b]\!]s' \\
[\![a\,b]\!] & = & \lambda s.\texttt{match}\ \ [\![a]\!]s\ \ \texttt{with}\ \ (x_a,s') \to \\
 & & \quad \texttt{match}\ \ [\![b]\!]s'\ \ \texttt{with}\ \ (x_b,s'') \to x_a\,x_b\,s''
\end{array}
$$

Exactly the same transformation but with different constructions

Outline

Commonalities of program transformations

Let us temporary abandon Haskell and return to pseudo-OCaml syntax

Consider the conversions to exception-returning style, state-passing style, and continuation-passing style. For constants, variables and λ-abstractions (ie., *values*), we have:

Pure	Exceptions	State	Continuations
$[\![N]\!]$	$= \quad Val(N)$	$= \quad \lambda s.(N, s)$	$= \quad \lambda k.kN$
$[\![x]\!]$	$= \quad Val(x)$	$= \quad \lambda s.(x, s)$	$= \quad \lambda k.kx$
$[\![\lambda x.a]\!]$	$= \quad Val(\lambda x.[\![a]\!])$	$= \quad \lambda s.(\lambda x.[\![a]\!], s)$	$= \quad \lambda k.k(\lambda x.[\![a]\!])$

In all three cases we **`return`** the values N, x, or $\lambda x.[\![a]\!]$ wrapped in some appropriate context.

Commonalities of program transformations

For `let` bindings we have

$$
\begin{aligned}
[\![\texttt{let}\ x = a\ \texttt{in}\ b]\!] &= \texttt{match}\ [\![a]\!]\ \texttt{with}\ \mathit{Exn}(z) \to \mathit{Exn}(z) \mid \mathit{Val}(x) \to [\![b]\!] \\
[\![\texttt{let}\ x = a\ \texttt{in}\ b]\!] &= \lambda s.\texttt{match}\ [\![a]\!]s\ \texttt{with}\ (x, s') \to [\![b]\!]s' \\
[\![\texttt{let}\ x = a\ \texttt{in}\ b]\!] &= \lambda k.[\![a]\!](\lambda x.[\![b]\!]k)
\end{aligned}
$$

In all three cases we extract the value resulting from the computation $[\![a]\!]$, we **`bind`** it to the variable x and proceed with the computation $[\![b]\!]$.

Commonalities of program transformations

For applications we have

$$
\begin{array}{rcl}
[\![a\,b]\!] & = & \texttt{match}\ [\![a]\!]\ \texttt{with} \\
& & \mid Exn(x_a) \to Exn(x_a) \\
& & \mid Val(x_a) \to \texttt{match}\ [\![b]\!]\ \texttt{with} \\
& & \qquad\qquad \mid Exn(y_b) \to Exn(y_b) \\
& & \qquad\qquad \mid Val(y_b) \to x_a\,y_b \\
[\![a\,b]\!] & = & \lambda s.\texttt{match}\ [\![a]\!]s\ \texttt{with}\ (x_a, s') \to \\
& & \quad \texttt{match}\ [\![b]\!]s'\ \texttt{with}\ (y_b, s'') \to x_a\,y_b\,s'' \\
[\![a\,b]\!] & = & \lambda k.[\![a]\!](\lambda x_a.[\![b]\!](\lambda y_b.x_a\,y_b\,k))
\end{array}
$$

We **bind** the value of $[\![a]\!]$ to the variable x_a, then **bind** the value of $[\![b]\!]$ to the variable y_b, then perform the application $x_a y_b$, and rewrap the result as needed.

Commonalities of program transformations

For types notice that if $a : \tau$ then $[\![a]\!] : [\![\tau]\!]$ `mon`
where
- $[\![\tau_1 \to \tau_2]\!] = \tau_1 \to [\![\tau_2]\!]$ `mon`
- $[\![B]\!] = B$ for bases types B.

For exceptions:

```
type α mon = Val of α | Exn of exn
```

For states:

```
type α mon  = state → α × state
```

For continuations:

```
type α mon = (α → answer) → answer
```

Monadic translation

The previous three translations are instances of the following translation

$$\begin{array}{rcl}
[\![N]\!] & = & \texttt{return}\ N \\
[\![x]\!] & = & \texttt{return}\ x \\
[\![\lambda x.a]\!] & = & \texttt{return}\ (\lambda x.[\![a]\!]) \\
[\![\texttt{let}\ x = a\ \texttt{in}\ b]\!] & = & [\![a]\!]\ \texttt{>>=}\ (\lambda x.[\![b]\!]) \\
[\![ab]\!] & = & [\![a]\!]\ \texttt{>>=}\ (\lambda x_a.[\![b]\!]\ \texttt{>>=}\ (\lambda y_b.x_a y_b))
\end{array}$$

just the monad changes, that is, the definitions of `bind` and `return`).

Exception monad

So the previous translation coincides with our exception returning transformation for the following definitions of bind and return:

```
type α mon = Val of α | Exn of exn

return a   = Val(a)

m >>= f    = match m with Exn(x) -> Exn(x) | Val(x) -> f x
```

bind encapsulates the propagation of exceptions in compound expressions such as the application *ab* or let bindings. As usual we have:

```
return : α → α mon

(>>=)  : α mon → (α → β mon) → β mon
```

Additional operations in this monad:

```
raise x = Exn(x)

trywith m f = match m with Exn(x) -> f x | Val(x) -> Val(x)
```

The State monad

To have the state-passing transformation we use instead the following definitions for return and bind:

```
type α mon = state → α × state

return a = λs. (a, s)

m >>= f = λs. match m s with (x, s’) -> f x s’
```

bind encapsulates the threading of the state in compound expressions.
Additional operations in this monad:

```
       ref x = λs. store_alloc x s

     deref r = λs. (store_read r s, s)

  assign r x = λs. store_write r x s
```

The Continuation monad

Finally the following monad instance yields the continuation-passing transformation:

```
type α mon = (α → answer) → answer

return a = λk. k a

m >>= f = λk. m (λv. f v k)
```

Additional operations in this monad:

```
callcc f = λk. f k k

throw x y = λk. x y
```

More on monadic translation

We can extend the monadic translation to more constructions of the language.

$$\begin{array}{rcl}
[\![\mu f.\lambda x.a]\!] & = & \mathtt{return}(\mu f.\lambda x.[\![a]\!]) \\
[\![a \;\mathbf{op}\; b]\!] & = & [\![a]\!] \;\mathtt{>>=}\; (\lambda x_a.[\![b]\!] \;\mathtt{>>=}\; (\lambda y_b.\mathtt{return}(x_a \;\mathbf{op}\; y_b))) \\
[\![C(a_1,...,a_n)]\!] & = & [\![a_1]\!] \;\mathtt{>>=}\; (\lambda x_1.\dots[\![a_n]\!] \;\mathtt{>>=}\; (\lambda x_n.\mathtt{return}(C(x_1,\dots,x_n)) \\
[\![\mathtt{match}\; a \;\mathtt{with}\; ..p..]\!] & = & [\![a]\!] \;\mathtt{>>=}\; (\lambda x_a.\mathtt{match}\; x_a \;\mathtt{with}\; ..[\![p]\!]...) \\
 & & \text{where } [\![C(x_1,...,x_n) \to a]\!] = C(x_1,...,x_n) \to [\![a]\!]
\end{array}$$

All these are parametric in the definition of bind and return.

Correctness of the monadic translation

The fundamental property of the monadic translation is that it does not alter the semantics of the computation it encodes. It just adds to the computation some effects.

Theorem

If $a \Rightarrow v$, then $[\![a]\!] \equiv \texttt{return}\ v'$

$$\textit{where } v' = \begin{cases} N & \textit{if } v = N \\ \lambda x.[\![a]\!] & \textit{if } v = \lambda x.a \end{cases}$$

Examples of monadic translation

```
[[ 1 + f x ]]  =
      (return 1) >>= (λx_1.
      ((return f) >>= (λx_2.
                  (return x) >>= (λx_3. x_2 x_3))) >>=( λx_4.
      return (x_1 + x_4)))
```

After administrative reductions using the first monadic law:

(`return x >>= f` is equivalent to `f x`)

```
[[ 1 + f x ]]  =
      (f x) >>= (λx_4. return (1 + x_4))
```

A second example

```
[[ μfact. λn. if n = 0 then 1 else n * fact(n-1) ]]  =
      return (μfact. λn.
                if n = 0
                then return 1
                else (fact(n-1)) >>= (λv. return (n * v))
             )
```

Summary

What we have done:

1. Take a program that performs some computation
2. Apply the monadic transformation to it. This yields a new program that uses `return` and >>= in it.
3. Choose a monad (that is, choose a definition for `return` and >>=) and the new programs embeds the computation in the corresponding monad (side-effects, exceptions, etc.)
4. You can now add in the program the operations specific to the chosen monad: although it includes effects the program is still *pure*.

Outline

Monads as a general programming technique

Monads provide a systematic way to *structure* programs into two well-separated parts:

- the proper algorithms, and
- the “plumbing” needed by *computation* of these algorithms to produce effects (state passing, exception handling, non-deterministic choice, etc).

In addition, monads can also be used to *modularize* code and offer new possibilities for reuse:

- Code in monadic form can be parametrized over a monad and reused with several monads.
- Monads themselves can be built in an incremental manner.

Back to Haskell

Let us put all this at work by writing in Haskell the canonical, efficient interpreter that ended our refresher course on operational semantics.

The canonical, efficient interpreter in OCaml (reminder)

```
# type term = Const of int | Var of int | Lam of term
            | App of term * term | Plus of term * term
   and value = Vint of int | Vclos of term * environment
   and environment = value list                      (* use Vec instead *)

# exception Error

# let rec eval e a =                       (* : environment -> term -> value *)
    match a with
    | Const n -> Vint n
    | Var n -> List.nth e n
    | Lam a -> Vclos(Lam a, e)
    | App(a, b) -> ( match eval e a with
        | Vclos(Lam c, e') ->
            let v = eval e b in
            eval (v :: e') c
        | _ -> raise Error)
    | Plus(a,b) ->  match (eval e a, eval e b) with
        | (Vint n, Vint m) -> Vint (n+m)
        |  _ -> raise Error

# eval [] (Plus(Const(5),(App(Lam(Var 0),Const(2)))));; (* 5+((λx.x)2)→7 *
- : value = Vint 7
```

Note: a `Plus` operator added

The canonical, efficient interpreter in Haskell

```
data Exp   = Const Integer                                    -- expressions
           | Var Integer
           | Plus Exp Exp
           | Abs Exp
           | App Exp Exp
data Value = Vint Integer                                     -- values
           | Vclos Env Exp

type Env   = [Value]                                          -- list of values

eval0 :: Env -> Exp -> Value
eval0 env (Const i )     = Vint i
eval0 env (Var n)        = env !! n                            -- n-th element
eval0 env (Plus e1 e2 ) = let Vint i1 = eval0 env e1
                              Vint i2 = eval0 env e2           -- let syntax
                          in Vint (i1 + i2 )
eval0 env (Abs e)        = Vclos env e
eval0 env (App e1 e2 )  = let Vclos env0 body  = eval0 env e1
                              val = eval0 env e2
                          in eval0 (val : env0) body
```

No exceptions: pattern matching may fail.

```
*Main> eval0 [] (App (Const 3) (Const 4))
*** Irrefutable pattern failed for pattern Main.Vclos env body
```

Haskell “do” Notation

Haskell has a very handy notation for monads

In a do block you can macro expand every intermediate line of the form

pattern <- *expression*   into   *expression* >>= \ *pattern* ->

and every intermediate line of the form

expression   into   *expression* >>= \ _ ->

This allows us to simplify the monadic translation for expressions which in Haskell syntax is defined as

$$\begin{array}{rcl}
[\![N]\!] & = & \texttt{return}\ N \\
[\![x]\!] & = & \texttt{return}\ x \\
[\![\lambda x.a]\!] & = & \texttt{return}\ (\backslash x\texttt{->}[\![a]\!]) \\
[\![\texttt{let}\ x = a\ \texttt{in}\ b]\!] & = & [\![a]\!]\ \texttt{>>=}\ (\backslash x\texttt{->}[\![b]\!]) \\
[\![a\,b]\!] & = & [\![a]\!]\ \texttt{>>=}\ (\backslash x_a\texttt{->}[\![b]\!]\ \texttt{>>=}\ (\backslash y_b\texttt{->}x_a y_b))
\end{array}$$

By using the do notation the last two cases become far simpler to understand

Monadic transformation in Haskell

$$
\begin{array}{rcl}
[\![N]\!] & = & \texttt{return}\ N \\
[\![x]\!] & = & \texttt{return}\ x \\
[\![\lambda x.a]\!] & = & \texttt{return}\ (\texttt{\\}x\texttt{->}[\![a]\!]) \\
[\![\texttt{let}\ x = a\ \texttt{in}\ b]\!] & = & \texttt{do}\ \ x\ \texttt{<-}\ [\![a]\!] \\
 & & \qquad [\![b]\!] \\
[\![a\,b]\!] & = & \texttt{do}\ \ x_a\ \texttt{<-}\ [\![a]\!] \\
 & & \qquad y_b\ \texttt{<-}\ [\![b]\!] \\
 & & \qquad x_a\,y_b
\end{array}
$$

The translation shows that `do` is the monadic version of `let`.

Monad at work

Let us apply the transformation to our canonical efficient interpreter

The canonical, efficient interpreter in monadic form

```
newtype Identity a = MkId a

instance Monad Identity where
    return a        = MkId a              -- i.e. return = id
    (MkId x) >>= f = f x                  -- i.e. x >>= f = f x

eval1 :: Env -> Exp -> Identity Value
eval1 env (Const i )     = return (Vint i)
eval1 env (Var n)        = return (env !! n)
eval1 env (Plus e1 e2 ) = do Vint i1  <- eval1 env e1
                             Vint i2  <- eval1 env e2
                             return (Vint (i1 + i2 ))
eval1 env (Abs e)        = return (Vclos env e)
eval1 env (App e1 e2 )  = do Vclos env0 body  <- eval1 env e1
                             val  <- eval1 env e2
                             eval1 (val : env0 ) body
```

We just replaced "`do`" for "`let`", replaced "`<-`" for "`=`", and put "`return`" in front of every value returned. Let us try to execute $(\lambda x.(x+1))4$

```
*Main> let MkId x = (eval1 [] (App(Abs(Plus(Var 0)(Const 1)))(Const 4)))
       in x
Vint 5
```

Although we wrote `eval1` for the Identity monad, the type of eval1 could be generalized to

```
eval1 :: Monad m => Env -> Exp -> m Value,
```

because we do not use any monadic operations other than `return` and `>>=` (hidden in the `do` notation): no `raise`, `assign`, `trywith`, ...
Recall that the type

```
Monad m => Env -> Exp -> m Value,
```

reads “for every type (constructor) `m` that is an instance of the type class `Monad`, the function has type `Env -> Exp -> m Value`”.
In our first definition of `eval1` we explicitly instantiated `m` into the Identity monad, but we can let the system instantiate it. For instance, if we give `eval` the generalized type above, then we do not need to extract the value encapsulated in the effect:

```
*Main>  (eval1 [] (App(Abs(Plus(Var 0)(Const 1)))(Const 4)))
Vint 5
```

The `ghci` prompt has run the expression in (ie, instantiated `m` by) the IO monad, because internally the interpreter uses the print function, which lives in just this monad.

Instantiating `eval` with the `Exception` monad

We decide to instantiate `m` in `eval` with the following monad:

```
data Exception e a = Val a | Exn e

instance Monad (Exception e) where
    return x    = Val x
    m >>= f     = case m of
                     Exn x -> Exn x
                     Val x -> f x

raise :: e -> Exception e a
raise x = Exn x

trywith :: Exception e a -> (e -> Exception e a) -> Exception e a
trywith m f = case m of
                 Exn x -> f x
                 Val x -> Val x
```

Note: Haskell provides an `Error` monad for exceptions. Not dealt with here.

Instantiating `eval` with the `Exception` monad

We can do dull instantiation:

```
eval1 :: Env -> Exp -> Exception e Value
eval1 env (Const i )    = return (Vint i)
eval1 env (Var n)       = return (env !! n)
eval1 env (Plus e1 e2 ) = do Vint i1  <- eval1 env e1
                             Vint i2  <- eval1 env e2
                             return (Vint (i1 + i2))
eval1 env (Abs e)       = return (Vclos env e)
eval1 env (App e1 e2 )  = do Vclos env0 body  <- eval1 env e1
                             val <- eval1 env e2
                             eval1 (val : env0) body
```

Not interesting since all we obtained is to encapsulate the result into a `Val` constructor.

The smart way

Use the exception monad to do as the OCaml implementation and raise an error when the applications are not well-typed

Instantiating eval with the Exception monad

New interpreter with exceptions:

```
eval2 :: Env -> Exp -> Exception String Value   -- exceptions as strings
eval2 env (Const i )     = return (Vint i)
eval2 env (Var n)        = return (env !! n)
eval2 env (Plus e1 e2 ) = do x1 <- eval2 env e1
                              x2 <- eval2 env e2
                              case (x1 , x2) of
                                 (Vint i1, Vint i2)
                                   -> return (Vint (i1 + i2))
                                 _ -> raise "type error in addition"
eval2 env (Abs e)        = return (Vclos env e)
eval2 env (App e1 e2 )  = do fun <- eval2 env e1
                              val <- eval2 env e2
                              case fun of
                                 Vclos env0 body
                                   -> eval2 (val : env0) body
                                 _ -> raise "type error in application"
```

And we see that the exception is correctly raised

```
*Main>  let Val x = ( eval2 [] (App (Abs (Var 0)) (Const 3)) ) in x
Vint 3
*Main>  let Exn x = ( eval2 [] (App (Const 2) (Const 3)) ) in x
"type error in application"
```

Instantiating `eval` with the `Exception` monad

Advantages:

- The function `eval2` is *pure*!
- Module few syntactic differences the code is really the same as code that would be written in an impure language (*cf*. the corresponding OCaml code)
- All “plumbing” necessary to preserve purity is defined separately (eg, in the `Exception` monad and its extra functions)
- In most cases the programmer does not even need to define “plumbing” since monads provided by standard Haskell libraries are largely sufficient.

A second try

Let us instantiate the type `Monad m => Env -> Exp -> m Value` with a different monad `m`. For our next example we choose the `State` monad.

Instantiating eval with the State monad

Goal: Add profiling capabilities by recording the number of evaluation steps.

```
-- The State Monad
data State s a = MkSt (s -> (a,s))

instance Monad (State s) where
    return a       = MkSt (\s -> (a,s))
    (MkSt g) >>= f = MkSt (\s -> let (v,s') = g s
                                     MkSt h = f v
                                  in h s')

get :: State s s
get = MkSt (\s -> (s,s))

put :: s -> State s ()
put s = MkSt (\_ -> ((),s))
```

To count evaluation steps we use an Integer number as state (ie, we use the `State Integer` monad). The operation `tick`, retrieves the hidden state from the computation, increases it and stores it back

```
tick :: State Integer ()
tick = do st <- get
          put (st + 1)
```

Instantiating `eval` with the `State` monad

```
eval3 :: Env -> Exp -> State Integer Value
eval3 env (Const i )     = do tick
                              return (Vint i)
eval3 env (Var n)        = do tick
                              return (env !! n)
eval3 env (Plus e1 e2 ) = do tick
                              x1 <- eval3 env e1
                              x2 <- eval3 env e2
                              case (x1 , x2) of
                                 (Vint i1, Vint i2)
                                    -> return (Vint (i1 + i2 ))
eval3 env (Abs e)        = do tick
                              return (Vclos env e)
eval3 env (App e1 e2 )  = do tick
                              fun <- eval3 env e1
                              val <- eval3 env e2
                              case fun of
                                 Vclos env0 body
                                    -> eval3 (val : env0 ) body
```

The evaluation of $(\lambda x.x)3$ takes 4 steps of reduction. This is shown by giving 0 as initial value of the state:

```
*Main>  let MkSt s = eval3 [] (App (Abs (Var 0)) (Const 3)) in s 0
(Vint 3,4)
```

Combining monads the hard way

What if we want *both* exceptions and state in our interpreter?

- Merging the code of `eval2` and `eval3` is straightforward: just add the code of `eval2` that raises the type-error exceptions at the end of the `Plus` and `App` cases in the definition of `eval3`.
- The problem is how to define the monad that supports both effects.

We can *write from scratch* the monad `m` that supports both effects.

```
eval4 :: Monad m => Env -> Exp -> m Value
```

Where the monad `m` above is one of the following two cases:

1. Use `StateOfException s e` for `m`: (with `s=Integer` and `e=[Char]`)

   ```
   data StateOfException s e a = State (s -> Exception e (s,a))
   ```

   the computation can either return a new pair state, value or generate an error (ie, when an exception is raised the state is discarded)
2. Use `ExceptionOfState s e` for `m`: (with `s=Integer` and `e=[Char]`)

   ```
   data ExceptionOfState s e a = State (s -> ((Exception e a), s ))
   ```

   the computation always returns a pair value and new state, and the value in this pair can be either an error or a normal value.

Combining monads the hard way

Notice that for the case `State (s -> ((Exception e a), s ))` there are two further possibilities, according to the state we return when an exception is caught. Each possibility corresponds to a different definition of `trywith`

1. backtrack the modifications made by the computation `m` that raised the exception:

```
  trywith m f = \s -> case m s of
                          (Val x , s') -> (Val x , s')
                          (Exn x , s') -> f x s
```

2. keep the modifications made by the computation `m` that raised the exception:

```
  trywith m f = \s -> case m s of
                          (Val x , s') -> (Val x , s')
                          (Exn x , s') -> f x s'
```

Avoid the boilerplate

Each of the standard monads is specialised to do exactly one thing. In real code, we often need several effects at once. Composing monads by hand or rewriting them from scratch soon reaches its limits

Combining monads by **compositionality**

By applying the monadic transformation to `eval` we passed from a function of type

```
Env -> Exp -> Value,
```

to a function of type

```
Monad m => Env -> Exp -> m Value,
```

In this way we made the code for `eval` parametric in the monad `m`.
Later we chose to instantiate `m` to some particular monad in order to use the specific characteristicts
IDEA: transform the code of an `instance` definition of the monad class so that this definition becomes parametric in some other monad `m`.

Monad transformer

A monad instance that is parametric in another monad is a *monad transformer*.

To work on the monad parameter, apply the monadic transformation to the definitions of instances

Monad Transformers

Monad Transformers can help:

- A monad transformer transforms a monad by adding support for an additional effect.
- A library of monad transformers can be developed, each adding a specific effect (state, error, . . .), allowing the programmer to mix and match.
- A form of *aspect-oriented programming*.

Monad Transformers

Monad Transformation in Haskell

- A *monad transformer* maps monads to monads. Represented by a type constructor `T` of the following kind:

```
T ::  (* -> *) -> (* -> *)
```

- Additionally, a monad transformer adds computational effects. A mapping `lift` from computations in the underlying monad to computations in the transformed monad is needed:

```
lift ::  M a ->  T M  a
```

Are you lost? ... Let us recap

Goal: write the following code where all the **plumbing** to handle effects is hidden in the definition of `m`

```
eval :: (Monad m) => Env -> Exp -> m Value

eval env (Const i )    = do tick
                            return (Vint i)
eval env (Var n)       = do tick
                            return (env !! n)
eval env (Plus e1 e2) = do tick
                            x1  <- eval env e1
                            x2  <- eval env e2
                            case (x1 , x2) of
                                 (Vint i1, Vint i2)
                                   -> return (Vint (i1 + i2 ))
                                 _ -> raise "type error in addition"
eval env (Abs e)       = do tick
                            return (Vclos env e)
eval env (App e1 e2)  = do tick
                            fun <- eval env e1
                            val <- eval env e2
                            case fun of
                                 Vclos env0 body
                                   -> eval (val : env0 ) body
                                 _ -> raise "type error in application"
```

Are you lost? ... Let us recap

The *dirty work* is in the definition of the monad `m` that will be used. Two ways are possible:

1. **Define `m` from scratch:** Define a new monad `m` so as it combines the effects of the Exception and of the State monads for which `raise` and `tick` are defined.
   Advantages: a fine control on the definition
   Drawbacks: no code reuse, hard to mantain and modify
2. **Define `m` by composition:** Define `m` by composing more elementary blocks that provide functionalities of *states* and *exceptions* respectively.
   Advantages: modular development; in many case it is possible to reuse components from the shelves.
   Drawbacks: Some trade-off since the building blocks may not provide exactly the sought combination of functionalities.

Monad transformers

We show the second technique by building the sought `m` from two *monad transformers* for exceptions and states respectively.

Step 1: defining the functionalities

We define two *subclasses* of the `Monad` class

EXCEPTION MONAD

An Exception Monad is a monad with an operation `raise` that takes a string and yields a monadic computation

```
class Monad m => ExMonad m where
  raise :: String -> m a
```

STATE MONAD

A State Monad is a monad with an operation `tick` that yields a computation on values of the unit type.

```
class Monad m => StMonad m where
  tick :: m ()
```

It is now possible to specify a type for `eval` so that its definition type-checks

```
eval :: (ExMonad m, StMonad m) => Env -> Exp -> m Value

eval env (Const i) = do tick
                          :
                      _ -> raise "type error in addition"
                          :
```

Step 2: defining the building blocks

We now need to define a monad `m` that is an instance of both `StMonad` and `ExMonad`.

We do it by composing two *monad transformers*

Definition (Monad transformer)

A *monad transformer* is a higher-order operator `t` that maps each monad `m` to a monad `(t m)`, equipped with an operation `lift` that promotes a computation `x :: m a` from the original monad `m` that is fed to `t`, to a computation

`(lift x) :: (t m) a`

on the monad `(t m)`.

Definition of the class of monad transformers

```
class MonadTrans t where
  lift :: Monad m => m a -> (t m) a
```

Example

If we want to apply to the monad `Exception String` a transformer `T` that provides some operation `xyz`, then we need to lift `raise` from `Exception String` to `T(Exception String)`.

Without the lifting the only operation defined for `T(Exception String)` would be `xyz`. With `lift` since

```
raise ::  String -> Exception String,
```

then:

```
lift.raise ::  String -> T(Exception String)
```

Nota bene

There is no magic formula to produce the transformer versions of a given monad

Step 2a: A monad transformer for exceptions

Consider again our first monad `Exception e`:

```
data Exception e a = Val a | Exn e

instance Monad (Exception e) where
    return x   = Val x
    m >>= f    = case m of Exn x -> Exn x ; Val x -> f x

raise :: e -> Exception e a
raise x = Exn x
```

We now want to modify the code above in order to obtain a transformer `ExceptionT` in which the computations are themselves on monads, that is:

```
data ExceptionT m a = MkExc (m (Exception String a))
```

The (binary) type constructor `ExceptionT` “puts exceptions inside” another monad `m` (convention: a monad transformers is usually named as the corresponding monad with a ’T’ at the end.)

We want `ExceptionT` to be a *monad transformer*, ie. `(ExceptionT m)` to be a monad: *we must define* `bind` *and* `return` *for the monad* `(ExceptionT m)`:

```
data ExceptionT m a = MkExc (m (Exception String a))

-- The 'recover' function just strips off the outer MkExc constructor,
-- for convenience
recover :: ExceptionT m a -> m (Exception String a)
recover (MkExc x) = x


-- return is easy. It just wraps the value first in the monad m
-- by return (of the underlying monad) and then in MkExc
returnET :: (Monad m) => a -> ExceptionT m a
returnET x = MkExc (return (Val x))


-- A first version for bind uses do and return to work on the
-- underlying monad m ... whatever it is.
bindET :: (Monad m) => (ExceptionT m a) -> ( a -> ExceptionT m b)
                                        ->  ExceptionT m b
bindET (MkExc x) f =                -- x of type m (Exception String a)
              MkExc (               -- we wrap the result in MkExc
                      do y <- x     -- y is of type Exception String a
                         case y of
                           Val z -> recover (f z)
                           Exn z -> return (Exn z) )
```

Notice the use of the monadic syntax (`do`, `return`,...) to work on the monad parameter `m`.

Step 2a: A monad transformer for exceptions

More compactly:

```
instance Monad m => Monad (ExceptionT m) where
    return x = MkExc (return (Val x))
    x >>= f  = MkExc (recover x >>= r)
                where r (Exn y) = return (Exn y)
                      r (Val y) = recover (f y)
```

Moreover, `(ExceptionT m)` is an exception monad, not just a plain one...

```
instance Monad m => ExMonad (ExceptionT m) where
  raise e = MkExc (return (Exn e))
```

`ExceptionT` is a monad tranformer because we can lift any action in `m` to an action in `(ExceptionT m)` by wrapping its result in a ' `Val`' constructor...

```
instance MonadTrans ExceptionT where
    lift g = MkExc $ do { x <- g; return (Val x) }
```

We can now use the `lift` operation to make `(ExceptionT m)` into a state monad whenever `m` is one, by lifting `m`'s `tick` operation to `(ExceptionT m)`.

```
instance StMonad m => StMonad (ExceptionT m) where
  tick = lift tick
```

Step 2b: A monad transformer for states

```
newtype StateT m a = MkStt ( Int -> m (a,Int))

-- strip off the MkStt constructor
apply :: StateT m a -> Int -> m (a, Int)
apply (MkStt f) = f

-- if m is a monad, then StateT m is a monad
instance Monad m => Monad (StateT m) where
  return x = MkStt $ \s -> return (x,s)
  p >>= q  = MkStt $ \s -> do (x,s') <- apply p s
                              apply (q x) s'

-- StateT is a monad transformer
instance MonadTrans StateT where
  lift g = MkStt $ \s -> do  x <- g; return (x,s)

-- if m is a monad, then StateT m is not only a monad
-- but also a STATE MONAD
instance (Monad m) => StMonad (StateT m) where
  tick = MkStt $ \s ->  return ((), s+1)

-- use lift to promote StateT m to an exception monad
instance ExMonad m => ExMonad (StateT m) where
  raise e = lift (raise e)
```

Lost again? Let us recap this Step 2

In Step 2 we defined some monad trasformers of the form `XyzT`.

1. To be a "transformer" `XyzT` must map monads into monads. So if `m` is a monad (ie., it provides `bind` and `return`), then so must `(XyzT m)` be. So we define `bind` and `return` for `(XyzT m)` and use monadic notation to work on the generic `m`.
2. But `(XyzT m)` must not only provide `bind` and `return`, but also some operations typical of some class `Xyz`, subclass of the `Monad` class. So we define also these operations by declaring that `(XyzT m)` is an instance of `Xyz`.
3. This is not enough for `XyzT` to be a transformer. It must also provide a `lift` operation. By defining it we declare that `XyzT` is an instance of the class `MonadTrans`
4. Finally we can use the `lift` function to make `(XyzT m)` "inherit" the characteristics of `m`: so if `m` is an instance of some monadic subclass `Abc`, then we can make also `(XyzT m)` be a `Abc` monad simply by lifting (by composition with `lift`) all the operations specific of `Abc`.

Step 3: Putting it all together...

Just a matter of assembling the pieces.

Interestingly, though, there are TWO ways to combine our transformers to build a monad with exceptions and state:

1.
   ```
   evalStEx :: Env -> Exp -> StateT (ExceptionT Identity) Value
   evalStEx = eval
   ```

2.
   ```
   evalExSt ::  Env -> Exp -> ExceptionT (StateT Identity) Value
   evalExSt = eval
   ```

Note that `ExceptionT Identity` and `StateT Identity` are respectively the `Exception` and `State` monads defined before, modulo two modifications:

1. Values are further wrapped in an inner `MkId` constructor
2. To enhance readibility I used distinct names for the types and their constructors, for instance:
   ```
   newtype StateT m a = MkStt (Int -> m (a,Int))
   ```
   rather then
   ```
   newtype StateT m a = StateT (Int -> m (a,Int))
   ```
   as it is customary in the Haskell library

Order matters

At first glance, it appears that `evalExSt` and `evalStEx` do the same thing...

```
five  = (App(Abs(Plus(Var 0)(Const 1)))(Const 4))    --(λx.(x+1))4
wrong = (App(Abs(Plus(Var 0)(Const 1)))(Abs(Var 0))) --(λx.(x+1))(λy.y)

*Main> evalStEx [] five
Vint 5, count: 6

*Main> evalExSt [] five
Vint 5, count: 6
```

BUT ...

```
*Main> evalStEx [] wrong
exception: type error in addition

*Main> evalExSt [] wrong
exception: type error in addition, count: 6
```

- `StateT (ExceptionT Identity)` either returns a state or an exception
- `ExceptionT (StateT Identity)` always returns a state

I omitted the code to print the results of monadic computations. It can be found in the accompagnying code: `http://www.irif.fr/~gc/slides/evaluator.hs`

The `Continuation` monad

Computation type: Computations which can be interrupted and resumed.

Binding strategy: Binding a function to a monadic value creates a new continuation which uses the function as the continuation of the monadic computation.

Useful for: Complex control structures, error handling and creating co-routines.

From `haskell.org`:

Abuse of the Continuation monad can produce code that is impossible to understand and maintain.

Many algorithms which require continuations in other languages do not require them in Haskell, due to Haskell's lazy semantics.

The Continuation monad

```
newtype Cont r a = Cont ((a -> r) -> r)

app :: Cont r a -> ((a -> r) -> r)          -- remove the wrapping Cont
app (Cont f) = f

instance Monad (Cont r) where
  return a = Cont $ \k -> k a                                -- = λk.k a
  (Cont c) >>= f = Cont $ \k -> c (\a -> app (f a) k) -- = λk.c(λa.f a k)
```

`Cont r a` is a CPS computation that produces an intermediate result of type `a` within a CPS computation whose final result type is `r`.

The return function simply creates a continuation which passes the value on.

The `>>=` operator adds the bound function into the continuation chain.

```
class (Monad m) => MonadCont m where
    callCC :: ((a -> m b) -> m a) -> m a

instance MonadCont (Cont r) where
    callCC f = Cont (\k -> app (f (\a -> Cont (\_ -> k a))) k)
```

Essentially (i.e., without constructors) the definition above states:

  `callCC f = λk.fkk`

ie., `f` is like a value but with an extra parameter *k* bound to its current continuatior

No need to define throw since we can directly use the continuation by applying it to a value, as shown in the next example

```
bar :: Char -> String -> Cont r String
bar c s = do
  msg <- callCC $ \k -> do
    let s’ = c : s
    if (s’ == "hello") then k "They say hello." else return ()
    let s’’ = show s’
    return ("They appear to be saying " ++ s’’)
  return msg
```

When you call k with a value, the entire callCC call returns that value. In other words, k is a ’goto’ statement: k in our example pops the execution out to where you first called callCC, the msg <- callCC $ ... line: no more of the argument to callCC (the inner do-block) is executed. This is shown by two different executions, to which we pass the function print as continuation:

```
main = do
     app (bar ’h’ "ello") print
     app (bar ’h’ "llo.") print
```

Which once compiled and executed produces the following output

```
"They say hello."
"They appear to be saying \"hllo.\""
```

A simpler example is the following one which contains a useless line:

```
bar :: Cont r Int
bar = callCC $ \k -> do
  let n = 5
  k n
  return 25
```

`bar` will always return `5`, and never `25`, because we pop out of `bar` before getting to the `return 25` line.

Summary

Purity has advantages but effects are unavoidable.

- To have them both, effects must be explicitly programmed.
- In order to separate the definition of the algorithm from the definition of the plumbing that manages the effects it is possible to use a monad. The monad centralizes all the programming that concerns effects.
- Several effects may be necessary in the same program. One can define the corresponding monad by composing monad transformers. These are functions from monads to monads, each handling a specific effect.

However

- Putting code in monadic form is easy and can be done automatically, but there is no magic formula to define monads or even derive from given monads the corresponding trasformers
- Understanding monadic code is relatively straightforward but writing and debugging monads or monads transformers from scracth may be dreadful.

Suggestion

Use **existing** monads and monads trasformers as much as possible.

Outline

Monads and ML Functors

- Monads define the bind and return functions that are the core of the plumbing of effects
- Specific operations for effects such as `raise` and `tick` are provided by subclasses of Monads (eg, `StMonad`, `ExMonad`).
- Modular development is obtained by *monad transformers* which are functions from monads to (subclasses of) monads.

We can reproduce monads by modules and transformers by functors.

Signature for monads

The Caml module signature for a monad is:

```
module type MONAD = sig
    type α mon
    val return: α -> α mon
    val bind: α mon -> (α -> β mon) -> β mon
end
```

The Identity monad

The Identity monad is a trivial instance of this signature:

```
module Identity = struct
    type α mon = α
    let return x = x
    let bind m f = f m
end
```

Monad Transformers

Monad transformer for exceptions

```
module ExceptionT(M: MONAD) = struct
  type α outcome = Val of α | Exn of exn
  type α mon = (α outcome) M.mon
  let return x = M.return (Val x)
  let bind m f =
    M.bind m (function Exn e -> M.return (Exn e) | Val v -> f v)
  let lift x = M.bind x (fun v -> M.return (Val v))
  let raise e = M.return (Exn e)
  let trywith m f =
    M.bind m (function Exn e -> f e | Val v -> M.return (Val v))
end
```

Notice the lesser flexibility due to the lack of the (static) overloading (provided by Haskell's type-classes) which obliges us to specify whose `bind` and `return` we use.

Also the fact that the `ExceptionT` functor returns a module that is (1) a *monad* (2) an instance of the *exception monad*, and (3) a *transformer*, is lost in the definition of the functions exported by the module [(1) holds because of `bind` and `return`, (2) because of `raise` and `trywith`, and (3) because of `lift`]

Monad Transformers

Monad transformer for state

```
module StateT(M: MONAD) = struct
  type α mon = state -> (α * state) M.mon
  let return x = fun s -> M.return (x, s)
  let bind m f =
    fun s -> M.bind (m s) (fun (x, s’) -> f x s’)
  let lift m = fun s -> M.bind m (fun x -> M.return (x, s))
  let ref x = fun s -> M.return (store_alloc x s)
  let deref r = fun s -> M.return (store_read r s, s)
  let assign r x = fun s -> M.return (store_write r x s)
end
```

Using monad transformers

```
module State = StateT(Identity)

module StateAndException = struct
      include ExceptionT(State)
      let ref x = lift (State.ref x)
      let deref r = lift (State.deref r)
      let assign r x = lift (State.assign r x)
   end
```

This gives a type α mon = state $\rightarrow$ α outcome $\times$ state, i.e. state is preserved when raising exceptions. The other combination, StateT(ExceptionT(Identity)) gives α mon = state $\rightarrow$ (α $\times$ state) outcome, i.e. state is discarded when an exception is raised.

Exercise

Define the functor for continuation monad transformer.

```
module ContTransf(M: MONAD) = struct
  type α mon = (α -> answer M.mon) -> answer M.mon
  let return x =
  let bind m f =
  let lift m =

  let callcc f =
  let throw c x =
end
```

References

- Philip Wadler. Monads for functional Programming. In Advanced Functional Programming, Proceedings of the Baastad Spring School, Lecture Notes in Computer Science n. 925, Springer, 1995.
- Martin Grabmüller. Monad Transformers Step by Step, Unpublished draft. 2006 `http://www.grabmueller.de/martin/www/pub/`

Subtyping

Outline

Outline

Simply Typed λ-calculus

Syntax

$$
\begin{array}{llcll}
\textit{Types} & T & ::= & T \to T & \text{function types} \\
 & & & \texttt{Bool} \mid \texttt{Int} \mid \texttt{Real} \mid \ldots & \text{basic types} \\
\textit{Terms} & a,b & ::= & \texttt{true} \mid \texttt{false} \mid 1 \mid 2 \mid \ldots & \text{constants} \\
 & & \mid & x & \text{variable} \\
 & & \mid & a\,b & \text{application} \\
 & & \mid & \lambda x{:}T.a & \text{abstraction}
\end{array}
$$

Reduction

$$\textit{Contexts} \quad C[\,] \quad ::= \quad [\,] \;\mid\; a[\,] \;\mid\; [\,]a \;\mid\; \lambda x{:}T.[\,]$$

$$\textsc{Beta}\quad (\lambda x{:}T.a)b \longrightarrow a[b/x] \qquad\qquad \textsc{Context}\quad \frac{a \longrightarrow b}{C[a] \longrightarrow C[b]}$$

Type system

Typing

$$\textsc{Var}\quad \Gamma \vdash x : \Gamma(x) \qquad \xrightarrow{}\textsc{Intro}\ \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T} \qquad \xrightarrow{}\textsc{Elim}\ \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

(plus the typing rules for constants).

Theorem (Subject Reduction)

If $\Gamma \vdash a : T$ and $a \longrightarrow^* b$, then $\Gamma \vdash b : T$.

We will essentially focus on the subject reduction property (a.k.a. *type preservation*), though well-typed programs also satisfy *progress*:

Theorem (Progress)

If $\varnothing \vdash a : T$ and $a \not\longrightarrow$, then a is a value

where a value is either a constant or a lambda abstraction

$$v ::= \lambda x{:}T.a \mid \texttt{true} \mid \texttt{false} \mid 1 \mid 2 \mid \ldots$$

Type checking algorithm

The deduction system is *syntax directed* and satisfies the *subformula property*. As such it describes a deterministic algorithm.

```
let rec typecheck gamma = function
  | $x$ -> gamma($x$)                                   (* Var rule   *)
  | $\lambda x{:}T.a$ -> $T \rightarrow$ (typecheck (gamma, $x:T$) $a$)  (* Intro rule *)
  | $ab$ -> let $T_1 \rightarrow T_2$ = typecheck gamma $a$ in     (* Elim rule  *)
          let $T_3$ = typecheck gamma b in
            if $T_1$==$T_3$ then $T_2$ else fail
```

Exercise. *Write the* `typecheck` *function for the following definitions:*

```
type stype = Int | Bool | Arrow of stype * stype

type term =
   Num of int | BVal of bool | Var of string
 | Lam of string * stype * term | App of term * term

exception Error
```

Use `List.assoc` for environments.

Subtyping

The rule for application requires the argument of the function to be *exactly of the same type* as the domain of the function:

$$\frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}\ (\to\textsc{Elim})$$

So, for instance, we **cannot:**

- Apply a function of type `Int` $\to$ `Int` to an argument of type `Odd` even though every odd number is an integer number, too.
- If we have records, apply the function $\lambda x{:}\{\ell : \texttt{Int}\}.(3 + x.\ell)$ to a record of type $\{\ell : \texttt{Int}, \ell' : \texttt{Bool}\}$
- If we are in OOP, send a message defined for objects of the class `Persons` to an instance of the subclass `Students`.

Subtyping polymorphism

We need a kind of polymorphism different from the ML one (parametric polymorphism).

Subtyping relation

- Define a pre-order (*ie*, a reflexive and transitive binary relation) $\leqslant$ on types: $\leqslant \subset$ *Types* $\times$ *Types* (some literature uses the notation `<:`)
- This *subtyping relation* has two possible interpretations:

  Containment: If $S \leqslant T$, then every value of type S *is also* of type T.
  For instance an odd number *is also* an integer, a student *is also* a person.
  Sometimes called a "**is_a**" relation.

  Substitutability: If $S \leqslant T$, then every value of type S can be *safely* used where a value of type T is expected.
  Where "safely" means, without disrupting type preservation and progress.
- We'll see how each interpretation has a formal counterpart.

Subtyping for simply typed λ-calculus

- We suppose to have a predefined preorder $\mathcal{B} \subset \mathit{Basic} \times \mathit{Basic}$ for basic types (given by the language designer).

  For instance take the reflexive and transitive closure of $\{(\mathtt{Odd}, \mathtt{Int}), (\mathtt{Even}, \mathtt{Int}), (\mathtt{Int}, \mathtt{Real})\}$
- To extend it to function types, we resort to the sustitutability interpretation. We will try to deduce when we can safely replace a function of some type by a term of a different type

Subtyping of arrows: intuition

Problem

Determine for which type S we have $S \leqslant T_1 \to T_2$

Let $g : S$ and $f : T_1 \to T_2$. Let us follow the **substitutability interpretation:**

1. If $a : T_1$, then we can apply f to a. If $S \leqslant T_1 \to T_2$, then we can apply g to a, as well.
   $\Rightarrow$ g is a function, therefore $S = S_1 \to S_2$
2. If $a : T_1$, then $f(a)$ is well typed. If $S_1 \to S_2 \leqslant T_1 \to T_2$, then also $g(a)$ is well-typed. g expects arguments of type S_1 but a is of type T_1
   $\Rightarrow$ we can safely use T_1 where S_1 is expected, ie $T_1 \leqslant S_1$
3. $f(a) : T_2$, but since g returns results in S_2, then $g(a) : S_2$. If I use g where f is expected, then it must be safe to use S_2 results where T_2 results are expected
   $\Rightarrow$ $S_2 \leqslant T_2$ must hold.

Solution

$$S_1 \to S_2 \leqslant T_1 \to T_2 \quad \Leftrightarrow \quad T_1 \leqslant S_1 \wedge S_2 \leqslant T_2$$

Covariance and contravariance

$$S_1 \to S_2 \leqslant T_1 \to T_2 \quad \Leftrightarrow \quad T_1 \leqslant S_1 \wedge S_2 \leqslant T_2$$

Notice the different orientation of containment on domains and co-domains.

We say that the type constructor $\to$ is

- *covariant* on codomains, since it preserves the direction of the relation;
- *contravariant* on domains, since it reverses the direction of the relation.

Containment interpretation:

The *containment interpretation* yields exactly the same relation as obtained by the *substitutability interpretation*. For instance a function that maps integers to integers ...

- *is also* a function that maps integers to reals: it returns results in `Int` so they will be also in `Real`.

  `Int`$\to$`Int`$\leqslant$ `Int`$\to$`Real` (covariance of the codomains)
- *is also* a function that maps odds to integers: when fed with integers it returns integers, so will do the same when fed with odd numbers.

  `Int`$\to$`Int`$\leqslant$ `Odd`$\to$`Int` (contravariance of the codomains)

Subtyping deduction system

$$\textsc{Basic}\ \frac{(B_1,B_2)\in\mathcal{B}}{B_1\leqslant B_2} \qquad \textsc{Arrow}\ \frac{T_1\leqslant S_1 \qquad S_2\leqslant T_2}{S_1\to S_2\leqslant T_1\to T_2}$$

$$\textsc{Refl}\ \frac{}{T\leqslant T} \qquad \textsc{Trans}\ \frac{T_1\leqslant T_2 \qquad T_2\leqslant T_3}{T_1\leqslant T_3}$$

This system is neither *syntax directed* nor satisfies the *subformula* property

How do we define an algorithm to check the subtyping relation?

Theorem (Admissibility of Refl and Trans)

In the system composed just by the rules Arrow and Basic:
1) $T\leqslant T$ is provable for all types T
2) If $T_1\leqslant T_2$ and $T_2\leqslant T_3$ are provable, so is $T_1\leqslant T_3$.

The rules Refl and Trans are *admissible*

Type system

We defined the subtyping relation and we know how to decide it. How do we use it for typing our programs?

$$\textsc{Var}\quad \Gamma \vdash x : \Gamma(x) \qquad \to\textsc{Intro}\ \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T} \qquad \to\textsc{Elim}\ \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

$$\textsc{Subsumption}\ \frac{\Gamma \vdash a : S \qquad S \leqslant T}{\Gamma \vdash a : T}$$

This corresponds to the *containment relation*:

if $S \leqslant T$ and a is of type S then a *is also* of type T

Subject reduction: If $\Gamma \vdash a : T$ and $a \longrightarrow^* b$, then $\Gamma \vdash b : T$.
Progress property: If $\varnothing \vdash a : T$ and $a \not\longrightarrow$, then a is a value

Typing algorithm

Var
$$\Gamma \vdash x : \Gamma(x)$$

$\rightarrow$Intro
$$\frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \rightarrow T}$$

$\rightarrow$Elim$_\leqslant$
$$\frac{\Gamma \vdash_{\mathcal{A}} a : S \rightarrow T \qquad \Gamma \vdash_{\mathcal{A}} b : U \qquad U \leqslant S}{\Gamma \vdash_{\mathcal{A}} ab : T}$$

$\rightarrow$Elim
$$\frac{\Gamma \vdash a : S \rightarrow T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

Subsumption
$$\frac{\Gamma \vdash a : S \qquad S \leqslant T}{\Gamma \vdash a : T}$$

Subsumption makes the type system non-algorithmic:

- it is not *syntax directed*: subsumption can be applied whatever the term.
- it does not satisfy the *subformula property*: even if we know that we have to apply subsumption which T shall we choose?

How do we define the typechecking algorithm?

Typing algorithm

$$\textsc{Var}\quad \Gamma \vdash_{\mathcal{A}} x : \Gamma(x) \qquad \xrightarrow{}\textsc{Intro}\ \frac{\Gamma, x : S \vdash_{\mathcal{A}} a : T}{\Gamma \vdash_{\mathcal{A}} \lambda x{:}S.a : S \to T} \qquad \to\textsc{Elim}_{\leqslant}\ \frac{\Gamma \vdash_{\mathcal{A}} a : S \to T \qquad \Gamma \vdash_{\mathcal{A}} b : U \qquad U \leqslant S}{\Gamma \vdash_{\mathcal{A}} ab : T}$$

1. The system is algorithmic: it describes a typing algorithm (exercise: program `typecheck` and `subtype` by using the previous structures)
2. The system conforms the substitutability interpretation: we *use* an expression of a subtype U where a supertype S is expected (note "use" = elimination rule).

How do we relate the two systems?

For subtyping, admissibility ensured that the system and the algorithm prove the same judgements. Here it is no longer true. For instance:

$$\varnothing \vdash \lambda x{:}\mathtt{Int}.x : \mathtt{Odd} \to \mathtt{Real} \qquad \text{but} \qquad \varnothing \not\vdash_{\mathcal{A}} \lambda x{:}\mathtt{Int}.x : \mathtt{Odd} \to \mathtt{Real}.$$

This is expected: Algorithm = one type returned for each typable term.

Soundness and completeness of the typing algorithm

a is typable by $\vdash$ $\iff$ a is typable by $\vdash_{\mathcal{A}}$

$\Leftarrow$ = soundness

$\Rightarrow$ = completeness

Theorem (Soundness)

If $\Gamma \vdash_{\mathcal{A}} a : T$, then $\Gamma \vdash a : T$

Theorem (Completeness)

If $\Gamma \vdash a : T$, then $\Gamma \vdash_{\mathcal{A}} a : S$ with $S \leqslant T$

Minimum type and soundness

Corollary (Minimum type)

If $\Gamma \vdash_{\mathcal{A}} a : T$ *then* $T = \min\{S \mid \Gamma \vdash a : S\}$

Proof. Let $\mathcal{S} = \{S \mid \Gamma \vdash a : S\}$. Soundness ensures that $\mathcal{S}$ is not empty. Completeness states that T is a lower bound of $\mathcal{S}$. Minimality follows by using soundness once more.

The corollary above explains that the typing algorithm works with the minimum types of the terms. It keeps track of the best type information available

Theorem (Algorithmic subject reduction)

If $\Gamma \vdash_{\mathcal{A}} a : T$ *and* $a \longrightarrow^* b$, *then* $\Gamma \vdash_{\mathcal{A}} b : S$ *with* $S \leqslant T$.

The theorem above explains that the computation reduces the minimum type of a program. As such it increases the type information about it.

Summary for simply-typed λ-calculs + $\leqslant$

- The *containment* interpretation of the subtyping relation corresponds to the "logical" view of the type system embodied by subsumption.
- The *substitutability* interpretation of the subtyping relation corresponds to the "algorithmic" view of the type system.
- To *define* the type system one usually starts from the "logical" system, which is simpler since subtyping is concentrated in the subsumption rule
- To *implement* the type system one passes to the substitutability view. Subsumption is eliminated and the check of the subtyping relation is distributed in the places where values are used/consumed. This in general corresponds to embed subtype checking into elimination rules.
- The obtained algorithm works on the *minimum types* of the logical system
- Computation reduces the (algorithmic) type thus increasing type information (the result of a computation represents the best possible type information: it is the *singleton type* containing the result).
- The last point makes *dynamic dispatch* (aka, dynamic binding) meaningful.

Products I

Syntax

$$\begin{array}{llcll} \textit{Types} & T & ::= & \ldots \mid T \times T & \text{product types} \\ \textit{Terms} & a, b & ::= & \ldots & \\ & & \mid & (a, a) & \text{pair} \\ & & \mid & \pi_i(a) \quad {\scriptstyle (i=1,2)} & \text{projection} \end{array}$$

Reduction

$$\pi_i((a_1, a_2)) \longrightarrow a_i \qquad {\scriptstyle (i=1,2)}$$

Typing

$$\times\textsc{Intro}\ \frac{\Gamma \vdash a_1 : T_1 \qquad \Gamma \vdash a_2 : T_2}{\Gamma \vdash (a_1, a_2) : T_1 \times T_2} \qquad\qquad \times\textsc{Elim}_i\ \frac{\Gamma \vdash a : T_1 \times T_2}{\Gamma \vdash \pi_i(a) : T_i}\ {\scriptstyle (i=1,2)}$$

Products II

Subtyping

$$\textsc{Prod}\quad \frac{S_1 \leqslant T_1 \qquad S_2 \leqslant T_2}{S_1 \times S_2 \leqslant T_1 \times T_2}$$

Exercise: *Check whether the above rule is compatible with the containement and/or the substitutability interpretation of the subtyping relation.*

The subtyping rule above is also algorithmic. Similarly, for the typing rules there is no need to embed subtyping in the elimination rules since π_i is an operator that works on all products, not a particular one (*cf.* with the application of a function, which requires a particular domain).

Of course subject reduction and progress still hold.

Exercise: *Define values and reduction contexts for this extension.*

Records

Up to now subtyping rules « lift » the subtyping relation $\mathcal{B}$ on basic types to constructed types. But if $\mathcal{B}$ is the identity relation, so is the whole subtyping relation. Record subtyping is non-trivial even when $\mathcal{B}$ is the identity relation.

Syntax

$$
\begin{array}{llcll}
\textit{Types} & T & ::= & \dots \mid \{\ell : T, \dots, \ell : T\} & \text{record types} \\[1ex]
\textit{Terms} & a, b & ::= & \dots & \\
 & & \mid & \{\ell = a, \dots, \ell = a\} & \text{record} \\
 & & \mid & a.\ell & \text{field selection}
\end{array}
$$

Reduction

$$\{\dots, \ell = a, \dots\}.\ell \longrightarrow a$$

Typing

{}INTRO

$$\frac{\Gamma \vdash a_1 : T_1 \;\dots\; \Gamma \vdash a_n : T_n}{\Gamma \vdash \{\ell_1 = a_1, \dots, \ell_n = a_n\} : \{\ell_1 : T_1, \dots, \ell_n : T_n\}}$$

{}ELIM

$$\frac{\Gamma \vdash a : \{\dots, \ell : T, \dots\}}{\Gamma \vdash a.\ell : T}$$

Record Subtyping

To define subtyping we resort once more on the substitutability relation. A record is “used” by selecting one of its labels.

We can replace some record by a record of different type if in the latter we can select the same fields as in the former and their contents can substitute the respective contents in the former.

Subtyping

$$\textsc{Record}\quad \frac{S_1 \leqslant T_1 \;\dots\; S_n \leqslant T_n}{\{\ell_1{:}S_1, \dots, \ell_n{:}S_n, \dots, \ell_{n+k}{:}S_{n+k}\} \leqslant \{\ell_1{:}T_1, \dots, \ell_n{:}T_n\}}$$

Exercise. *Which are the algorithmic typing rules?*

Outline

Iso-recursive and Equi-recursive types

Lists are a classic example of recursive types:

$$X \quad \approx \quad (\mathtt{Int} \times X) \vee \mathtt{Nil}$$

also written as $\mu X.((\mathtt{Int} \times X) \vee \mathtt{Nil})$

Two different approaches according to whether $\approx$ is interpreted as an isomorphism or an equality:

Iso-recursive types: $\mu X.((\mathtt{Int} \times X) \vee \mathtt{Nil})$ is considered *isomorphic* to its one-step unfolding $(\mathtt{Int} \times \mu X.((\mathtt{Int} \times X) \vee \mathtt{Nil})) \vee \mathtt{Nil})$. Terms include a pair of built-in coercion functions for each recursive type $\mu X.T$:

$$\mathtt{unfold} \ : \mu X.T \to T[\mu X.T/X] \qquad \mathtt{fold} \ : T[\mu X.T/X] \to \mu X.T$$

Equi-recursive types: $\mu X.((\mathtt{Int} \times X) \vee \mathtt{Nil})$ is considered *equal* to its one-step unfolding $(\mathtt{Int} \times \mu X.((\mathtt{Int} \times X) \vee \mathtt{Nil})) \vee \mathtt{Nil})$. The two types are completely interchangeable. No support needed from terms.

Subtyping for recursive types generalizes the equi-recursive approach.
The $\approx$ relation corresponds to subtyping in both directions:

$$\mu X.T \leqslant T[\mu X.T/X] \qquad\qquad T[\mu X.T/X] \leqslant \mu X.T$$

Recursive types are weird

- To add (equi-)recursive types you do not need to add any new term
- You don't even need to have recursion on terms:

$$\mu X.((\mathtt{Int} \times X) \vee \mathtt{Nil})$$

interpret the type above as the *finite* lists of integers.

Then $\mu X.(\mathtt{Int} \times X)$ is the empty type.

- Actually if you have recursive terms and allow infinite values you can easily jeopardize decidability of the subtyping relation (which resorts to checking type emptiness)
- This contrasts with their intuition which looks simple: we always informally applied a rule such as:

$$\frac{A, X \leqslant Y \vdash S \leqslant T}{A \vdash \mu X.S \leqslant \mu Y.T}$$

Subtyping recursive types

Syntax

Types	T	$::=$	$\mathtt{Any}$	top type
		$\mid$	$T \rightarrow T$	function types
		$\mid$	$T \times T$	product types
		$\mid$	X	type variables
		$\mid$	$\mu X.T$	recursive types

where T is *contractive*, that is (two equivalent definitions):

1. T is contractive iff for every subexpression $\mu X.\mu X_1.\ldots.\mu X_n.S$ it holds $S \neq X$.
2. T is contractive iff every type variable X occurring in it is separated from its binder by a $\rightarrow$ or a $\times$.

Subtyping recursive types

The subtyping relation is defined *COINDUCTIVELY* by the rules

$$\textsc{Top}\ \frac{}{T \leqslant \mathtt{Any}} \qquad \textsc{Prod}\ \frac{S_1 \leqslant T_1 \qquad S_2 \leqslant T_2}{S_1 \times S_2 \leqslant T_1 \times T_2} \qquad \textsc{Arrow}\ \frac{T_1 \leqslant S_1 \qquad S_2 \leqslant T_2}{S_1 \to S_2 \leqslant T_1 \to T_2}$$

$$\textsc{Unfold Left}\ \frac{S[\mu X.S/X] \leqslant T}{\mu X.S \leqslant T} \qquad \textsc{Unfold Right}\ \frac{S \leqslant T[\mu X.T/X]}{S \leqslant \mu X.T}$$

Coinductive definition

1. Why coinduction?
2. Why no reflexivity/transitivity rules?
3. Why no rule to compare two μ-types?

Short answers (more detailed answers to come):

1. Because we compare infinite expansions
2. Because it would be unsound
3. Useless since obtained by coinduction and unfold

Example of coinductive derivation

$$
\textsc{Unfold Left}\ \dfrac{\textsc{Unfold Right}\ \dfrac{\textsc{Arrow}\ \dfrac{\texttt{Even} \leqslant \texttt{Int} \qquad \mu X.\texttt{Int} \to X \leqslant \mu Y.\texttt{Even} \to Y}{\texttt{Int} \to (\mu X.\texttt{Int} \to X) \leqslant \texttt{Even} \to (\mu Y.\texttt{Even} \to Y)}}{\texttt{Int} \to (\mu X.\texttt{Int} \to X) \leqslant \mu Y.\texttt{Even} \to Y}}{\mu X.\texttt{Int} \to X \leqslant \mu Y.\texttt{Even} \to Y}
$$

Notice the use of coinduction

Amadio and Cardelli's subtyping algorithm

Let $A \subset \mathit{Types} \times \mathit{Types}$

$$\frac{}{A \vdash S \leqslant T}\ (S,T) \in A$$

$$\frac{}{A \vdash S \leqslant \mathtt{Any}}\ (S,\mathtt{Any}) \notin A$$

$$\frac{A' \vdash S_1 \leqslant T_1 \qquad A' \vdash S_2 \leqslant T_2}{A \vdash S_1 \times S_2 \leqslant T_1 \times T_2}\ A' = A \cup (S_1 \times S_2, T_1 \times T_2); A \neq A'$$

$$\frac{A' \vdash T_1 \leqslant S_1 \qquad A' \vdash S_2 \leqslant T_2}{A \vdash S_1 \to S_2 \leqslant T_1 \to T_2}\ A' = A \cup (S_1 \to S_2, T_1 \to T_2); A \neq$$

$$\frac{A' \vdash S[\mu X.S/X] \leqslant T}{A \vdash \mu X.S \leqslant T}\ A' = A \cup (\mu X.S, T); A \neq A'; T \neq \mathtt{Any}$$

$$\frac{A' \vdash S \leqslant T[\mu X.T/X]}{A \vdash S \leqslant \mu X.T}\ A' = A \cup (S, \mu X.T); A \neq A'; S \neq \mu Y.U$$

Properties

Theorem (Soundness and Completeness)

Let S and T be closed types. $S \leqslant T$ belongs the relation coinductively defined by the rules in slide 374 if and only if $\varnothing \vdash S \leqslant T$ is provable

To see the proof of the above theorem you can refer to the following reference Pierce et al. Recursive types revealed, Journal of Functional Programming, 12(6):511-548, 2002.

Notice that the algorithm above is exponential. We will show how to define an $O(n^2)$ algorithm to decide $S \leqslant T$, where n is the total number of different subexpressions of $S \leqslant T$.

Induction and coinduction

Intuition

Given a deduction system, it characterizes two possible distinct sets (of provable judgements) according to whether an inductive or a coinductive approach is used.

Let $\mathcal{F}$ be a deduction system on a universe $\mathcal{U}$ (i.e. a monotone function from $\mathcal{P}(\mathcal{U})$ to $\mathcal{P}(\mathcal{U})$). A set $X \in \mathcal{P}(\mathcal{U})$ is:

$\mathcal{F}$-closed if it contains all the elements that can be deduced by $\mathcal{F}$ with hypothesis in X.

$\mathcal{F}$-consistent if every element of X can be deduced by $\mathcal{F}$ from other elements in X.

Induction and coinduction

A deduction system

- *inductively* defines the least $\mathcal{F}$-closed set
- *coinductively* defines the greatest $\mathcal{F}$-consistent set

Induction and coinduction

induction: start from $\varnothing$, add all the consequences of the deduction system, and iterate.

coinduction: start from $\mathcal{U}$, remove all elements that are not consequence of other elements, and iterate.

Observation

In all the (algorithimic, ie without refl and trans) subtyping system met so far, the two coincide. This is not true in general, due to the presence of *self-justifying sets*, that is sets in which the deductions do not start just by axioms.

Example:

$$\mathcal{U} = \{a, b, c, d, e, f, g\} \qquad \frac{a}{b} \quad \frac{b}{c} \quad \frac{c}{a} \quad \frac{}{d} \quad \frac{d}{e} \quad \frac{f}{g}$$

Inductively: $\{d, e\}$

Coinductively: $\{a, b, c, d, e\}$

Self-justifying set: $\{a, b, c\}$

Exercises

1. Let $\mathcal{U} = \mathbb{Z}$ and take as deduction system all the instances of the rule
$$\frac{n}{n+1}$$
for $n \in \mathbb{Z}$. Which are the sets inductively and coinductively defined by it?
2. Same question but with $\mathcal{U} = \mathbb{N}$.
3. Same question but with $\mathcal{U} = \mathbb{N}^2$ and as deduction system all the rules instance of
$$\frac{(m,n) \qquad (n,o)}{(m,o)}$$
for $m, n, o \in \mathbb{N}$

Why Coinduction for Recursive types?

We want to use $S = \mu X.\mathtt{Int} \to X$ where $T = \mu Y.\mathtt{Even} \to Y$ is expected.

Use the substitutability interpretation.

Let $e : T$ then e:

1. waits for an $\mathtt{Even}$ number,
2. fed by an $\mathtt{Even}$ number returns a function that behaves similarly: (1) wait for an $\mathtt{Even}$ ...

Now consider $f : S$, then f:

1. waits for an $\mathtt{Int}$ number,
2. fed by an $\mathtt{Int}$ (or a $\mathtt{Even}$) number returns a function that behaves similarly: (1) wait for ...

S and *T* are in subtyping relation because
their infinite expansions are in subtyping relation.

$$S \leqslant T \quad\Longrightarrow\quad \mathtt{Int} \to S \leqslant \mathtt{Even} \to T \quad\Longrightarrow\quad S \leqslant T \wedge \mathtt{Even} \leqslant \mathtt{Int}$$

This is exactly the proof we saw at the beginning:

$$\textsc{Unfold Left}\;\cfrac{\textsc{Unfold Right}\;\cfrac{\textsc{Arrow}\;\cfrac{\texttt{Even}\leqslant\texttt{Int} \qquad \overbrace{\mu X.\texttt{Int}\to X}^{S}\leqslant\overbrace{\mu Y.\texttt{Even}\to Y}^{T}}{\texttt{Int}\to(\mu X.\texttt{Int}\to X)\leqslant\texttt{Even}\to(\mu Y.\texttt{Even}\to Y)}}{\texttt{Int}\to(\mu X.\texttt{Int}\to X)\leqslant\mu Y.\texttt{Even}\to Y}}{\underbrace{\mu X.\texttt{Int}\to X}_{S}\leqslant\underbrace{\mu Y.\texttt{Even}\to Y}_{T}}$$

Coinduction

$S\leqslant T$ is not an axiom but $\{S\leqslant T\ ,\ \texttt{Even}\leqslant\texttt{Int}\}$ is a *self-justifying set*.

Observation:

1. The deduction above shows why a specific rule for μ is useless (apply consecutively the two unfold rules).
2. If we added reflexivity and/or transitivity rules, then $\mathcal{U}$ would be $\mathcal{F}$-consistent (*cf.* the third exercise few slides before).

A naive implementation of the Amadio-Cardelli algorithm is exponential (why?). If we “thread” the computation of the memoization environments we obtain a quadratic complexity. This is done as follows:

```
$subtype(A, S, T)$  =  if $(S, T) \in A$ then $A$ else
                       let $A_0 = A \cup \{(S, T)\}$ in
                       if $T = \texttt{Any}$ then $A_0$
                         else if $S = S_1 \times S_2$ and $T = T_1 \times T_2$ then
                             $subtype(subtype(A_0, S_1, T_1), S_2, T_2)$
                         else if $S = S_1 \to S_2$ and $T = T_1 \to T_2$ then
                             $subtype(subtype(A_0, T_1, S_1), S_2, T_2)$
                         else if $T = \mu X.T_1$ then
                             $subtype(A_0, S, T_1[\mu X.T_1/X])$
                         else if $S = \mu X.S_1$ then
                             $subtype(A_0, S_1[\mu X.S_1/X], T)$
                        else fail
```

Compare the previous algorithm with the Amadio-Cardelli algorithm:

$$\frac{}{A \vdash S \leqslant T}\ (S,T) \in A$$

$$\frac{}{A \vdash S \leqslant \texttt{Any}}\ (S,\texttt{Any}) \notin A$$

$$\frac{A' \vdash S_1 \leqslant T_1 \qquad A' \vdash S_2 \leqslant T_2}{A \vdash S_1 \times S_2 \leqslant T_1 \times T_2}\ A' = A \cup (S_1 \times S_2, T_1 \times T_2); A \neq A$$

$$\frac{A' \vdash T_1 \leqslant S_1 \qquad A' \vdash S_2 \leqslant T_2}{A \vdash S_1 \to S_2 \leqslant T_1 \to T_2}\ A' = A \cup (S_1 \to S_2, T_1 \to T_2); A \neq$$

$$\frac{A' \vdash S[\mu X.S/X] \leqslant T}{A \vdash \mu X.S \leqslant T}\ A' = A \cup (\mu X.S, T); A \neq A'; T \neq \texttt{Any}$$

$$\frac{A' \vdash S \leqslant T[\mu X.T/X]}{A \vdash S \leqslant \mu X.T}\ A' = A \cup (S,\mu X.T); A \neq A'; S \neq \mu Y.U$$

They both check containment in the relation coinductively defined by:

$$\textsc{Top}\ \frac{}{T \leqslant \texttt{Any}} \qquad \textsc{Prod}\ \frac{S_1 \leqslant T_1 \qquad S_2 \leqslant T_2}{S_1 \times S_2 \leqslant T_1 \times T_2} \qquad \textsc{Arrow}\ \frac{T_1 \leqslant S_1 \qquad S_2 \leqslant T_2}{S_1 \to S_2 \leqslant T_1 \to T_2}$$

$$\textsc{Unfold Left}\ \frac{S[\mu X.S/X] \leqslant T}{\mu X.S \leqslant T} \qquad \textsc{Unfold Right}\ \frac{S \leqslant T[\mu X.T/X]}{S \leqslant \mu X.T}$$

But the former is far more efficient.

Outline

References

- R. Amadio and L. Cardelli. Subtyping recursive types. ACM Transactions on Programming Languages and Systems, 14(4):575-631, 1993.
- Pierce et al. Recursive types revealed, Journal of Functional Programming, 12(6):511-548, 2002.

XML Programming

Outline

Outline

XML is just tree-structured data:

```
<biblio>
  <book status="available">
    <title>Object-Oriented Programming</title>
    <author>Giuseppe Castagna</author>
  </book>
  <book>
    <title>A Theory of Objects</title>
    <author>Martín Abadi</author>
    <author>Luca Cardelli</author>
  </book>
<biblio>
```

Types describe the set of valid documents

```
<?xml version="1.0"?>
  <!DOCTYPE biblio [
  <!ELEMENT biblio (book*)>
  <!ELEMENT book (title, (author|editor)+, price?)>
  <!ATTLIST book status (available|borrowed) #IMPLIED>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT editor (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
]>
```

Programming with XML

How to manipulate data that is in XML format in a programming language?

- Level 0: textual representation of XML documents
  - AWK, sed, Perl regexp
- Level 1: abstract view provided by a parser
  - SAX, DOM, . . .
- Level 2: untyped XML-specific languages
  - XSLT, XPath
- **Level 3: XML types taken seriously**
  - XDuce, Xtatic
  - XQuery
  - **CDuce**
  - C_ω (Microsoft)
  - . . .

Examples

Level 1: DOM in Javascript

Print the titles of the book in the bibliography

```
<script>
  xmlDoc=loadXMLDoc("biblio.xml");
  x=xmlDoc.getElementsByTagName("book");
  for (i=0;i<x.length;i++){
    document.write(x[i].childNodes[0].nodeValue);
    document.write("<br>");
  }
</script>
```

Level 2: XPath

The same in XPath:

```
/biblio/book/title
```

Select all titles of books whose price > 35

```
/biblio/book[price>35]/title
```

Level 2: XSLT

XSLT uses XPath to extract information (as a pattern in pattern matching)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
  <h2>Books Price List</h2>
  <table border="1">
    <tr bgcolor="#9acd32">
      <th>Title</th>
      <th>Price</th>
    </tr>
    <xsl:for-each select="biblio/book">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="price"/></td>
    </tr>
    </xsl:for-each>
  </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Types are ignored

- In DOM nothing ensures that the read of a next node suceeds
- In XPath `/biblio/title/book` return an empty set of nodes rather than a type error
- Likewise the use of wrong XPath expressions in XSLT is unnoticed and yields empty XML documents as result (in the previous example the fact that `price` is optional is not handled).

Level 3: Recent languages take types seriously

- XDuce, Xtatic
- XQuery
- **CDuce**
- C_ω
- ...

How to add XML types in programming languages?

We need *set-theoretic* type connectives

Outline

Set-theoretic types

We consider the following possibly recursive types:

$$\texttt{T} ::= \texttt{Bool} \mid \texttt{Int} \mid \texttt{Any} \mid \texttt{(T,T)} \mid \texttt{T} \vee \texttt{T} \mid \texttt{T \& T} \mid \texttt{not(T)} \mid \texttt{T-->T}$$

Useful for:

1. XML types
2. Precise typing of pattern matching
3. Overloaded functions
4. Mixins
5. General programming paradigms

Let us see each point more in detail

Note: henceforward I will sometimes use $\texttt{T}_1 | \texttt{T}_2$ to denote $\texttt{T}_1 \vee \texttt{T}_2$

1. XML types

```
<?xml version="1.0"?>
  <!DOCTYPE biblio [
  <!ELEMENT biblio (book*)>
  <!ELEMENT book (title, (author|editor)+, price?)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT editor (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
]>
```

Can be encoded with union and recursive types

```
type Biblio = (‘biblio,X)
type      X = (Book,X) ∨ ‘nil

type Book = (‘book,(Title, Y∨Z))
type    Y = (Author,Y∨(Price,‘nil)∨‘nil)
type    Z = (Editor,Z∨(Price,‘nil)∨‘nil)

type Title  = (‘title,String)
type Author = (‘author,String)
type Editor = (‘editor,String)
type Price  = (‘price,String)
```

2. Precise typing of pattern matching (I)

Consider the following pattern matching expression

$$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$$

where patterns are defined as follows:

$$p ::= x \mid (p\,,p) \mid p|p \mid p\&p$$

If we interpret types as set of values

$$t = \{v \mid v \text{ is a value of type } t\}$$

then the set of all values that match a pattern is a type

$$\{\!\!\{p\}\!\!\} = \{v \mid v \text{ is a value that matches } p\}$$

$$\begin{array}{rcl} \{\!\!\{x\}\!\!\} & = & \texttt{Any} \\ \{\!\!\{(p_1\,,p_2)\}\!\!\} & = & (\{\!\!\{p_1\}\!\!\}\,,\{\!\!\{p_2\}\!\!\}) \\ \{\!\!\{p_1|p_2\}\!\!\} & = & \{\!\!\{p_1\}\!\!\} \vee \{\!\!\{p_2\}\!\!\} \\ \{\!\!\{p_1\&p_2\}\!\!\} & = & \{\!\!\{p_1\}\!\!\}\ \&\ \{\!\!\{p_2\}\!\!\} \end{array}$$

2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching:*

$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$

Suppose that $e : \mathtt{T}$ and let us write $\mathtt{T}_1 \backslash \mathtt{T}_2$ for $\mathtt{T}_1 \,\&\, \mathtt{not}(\mathtt{T}_2)$

- To infer the type $\mathtt{T}_1$ of e_1 we need $\mathtt{T} \,\&\, \wr p_1 \int$;
- To infer the type $\mathtt{T}_2$ of e_2 we need $(\mathtt{T} \backslash \wr p_1 \int) \,\&\, \wr p_2 \int$;
- The type of the match expression is $\mathtt{T}_1 \vee T_2$.
- Pattern matching is exhaustive if $\mathtt{T} \leqslant \wr p_1 \int \vee \wr p_2 \int$;

Formally:

[MATCH]

$$\frac{\Gamma \vdash e : \mathtt{T} \qquad \Gamma, \mathtt{T} \,\&\, \wr p_1 \int / p_1 \vdash e_1 : \mathtt{T}_1 \qquad \Gamma, \mathtt{T} \backslash \wr p_1 \int / p_2 \vdash e_2 : \mathtt{T}_2}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p_1 \texttt{->} e_1\ \texttt{|}\ p_1 \texttt{->} e_2 : \mathtt{T}_1 \vee \mathtt{T}_2} (\mathtt{T} \leqslant \wr p_1 \int \vee \wr p_2 \int$$

where $\mathtt{T}/p$ is the type environment for the capture variables in p when the pattern is matched against values in $\mathtt{T}$.
(e.g., $((\mathtt{Int}\,,\mathtt{Int}) \vee (\mathtt{Bool}\,,\mathtt{Char}))/(x\,,y)$ is $x : \mathtt{Int} \vee \mathtt{Bool}, y : \mathtt{Int} \vee \mathtt{Char}$)

3. Overloaded functions

Intersection types are useful to type overloaded functions (in the Go language):

```
package main
import "fmt"
func Opposite (x interface{}) interface{} {
  var res interface{}
  switch value := x.(type) {
    case bool:
      res = (!value)            // x has type bool
    case int:
      res = (-value)            // x has type int
  }
  return res
}

func main() { fmt.Println(Opposite(3) , Opposite(true)) }
```

In Go `Opposite` has type `Any-->Any` (every value has type `interface{}`).
Better type with intersections `Opposite:  (Int-->Int) & (Bool-->Bool)`

Intersections can also to give a more refined description of standard functions:

```
func Successor(x int) { return(x+1) }
```

which could be typed as `Successor:(Odd-->Even) & (Even-->Odd)`

2+3. Precise typing of OCaml

Exercise:

1. What is the type returned by

   ```
   let foo = function
      | (‘A,‘B) -> true
      | (‘B,‘A) -> false
   ```

   and what is the problem ?

2. Which type could we give if we had full-fledged union types?

3. Give an intersection type that refines the previous type

4. Typing of Mixins

Intersection types are used in Microsoft's Typescript to type mixins.

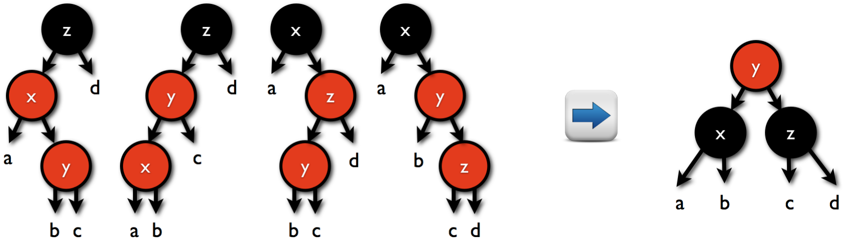
```
function extend<T, U>(first: T, second: U): T & U {
    /* <T> exp is a type cast (equivalent: exp as T) */
    let result = <T & U>{};
    for (let id in first) {
            (<any>result)[id] = (<any>first)[id]; }
    for (let id in second) { if (!result.hasOwnProperty(id)) {
            (<any>result)[id] = (<any>second)[id]; } }
    return result;
}
class Person {
    constructor(public name: string) { }
}
interface Loggable {
    log(): void;
}
class ConsoleLogger implements Loggable {
    log() { ... }
}

var jim = extend(new Person("Jim"), new ConsoleLogger());
var n = jim.name;
jim.log();
```

5. General programming paradigms

Consider red-black trees. Recall that they must satisfy 4 invariants.

1. the root of the tree is black
2. the leaves of the tree are black
3. no red node has a red child
4. every path from root to a leaf contains the same number of black nodes

The key of Okasaki's insertion is the function `balance` which transforms an *unbalanced tree*, into a *valid red-black tree* (as long as a, b, c, and d are valid):



In ML we need GADTs to enforce the invariants.

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree)|(RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree)|(RBtree,Wrong) )

let balance: (Unbal→Rtree) & ( (β\Unbal)→(β\Unbal) ) =
 function
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
      -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert: (α, Btree)→Btree =
 function ( x , t ) ->
  let ins: (Leaf→Rtree) & (Btree→RBtree\Leaf) & (Rtree→Rtree|Wrong) =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

Cutting edge research

Type checking the previous definitions is not so difficult.
The hard part is to type partial applications:

$$\texttt{map} : (\alpha \to \beta) \to [\,\alpha\,] \to [\,\beta\,]$$

$$\texttt{balance} : (\texttt{Unbal} \to \texttt{Rtree})\ \&\ ((\beta \setminus \texttt{Unbal}) \to (\beta \setminus \texttt{Unbal}))$$

$$\begin{aligned}\texttt{map balance} : &\ ([\,\texttt{Unbal}\,] \to [\,\texttt{Rtree}\,])\\ \&&\ ([\,\alpha \setminus \texttt{Unbal}\,] \to [\,\alpha \setminus \texttt{Unbal}\,])\\ \&&\ ([\,\alpha \mid \texttt{Unbal}\,] \to [(\alpha \setminus \texttt{Unbal}) \mid \texttt{Rtree}\,])\end{aligned}$$

Fortunately, programmers (and you) are spared from these gory details.

New languages use union and intersections

Facebook's Flow:

```
// @flow
function toStringPrimitives(val: number | boolean | string) {
  return String(val);
}



type One = { foo: number };
type Two = { bar: boolean };

type Both = One & Two;

var value: Both = {
  foo: 1,
  bar: true
};
```

New languages use union and intersections

Typed-Racket

```
(let ([a-number 37])
    (if (even? a-number)
        'yes
        'no))
- : Symbol [more precisely: (U 'no 'yes)]
'no
```

```
(: f : (case-> (-> True Integer Integer)
               (-> False Boolean Boolean)))
  (define (f condition x)
    (if condition
       (add1 x)
       (not x)))
```

How to understand/explain set-theoretic type connectives?

- The type connectives union, intersection, and negation are completely defined by the subtyping relation:
    - $\mathtt{T}_1 \vee \mathtt{T}_2$ is the least upper bound of $\mathtt{T}_1$ and $\mathtt{T}_2$
    - $\mathtt{T}_1 \mathbin{\&} \mathtt{T}_2$ is the greatest lower bound of $\mathtt{T}_1$ and $\mathtt{T}_2$
    - `not(`T`)` is the only type whose union and intersection with `T` yield the `Any` and `Empty` types, respectively.
- Defining (and deciding) subtyping for *type connectives* (i.e., $\vee$, $\&$, `not()`) is far more difficult than for *type constructors* (i.e., `-->`, $\times$, $\{...\}$, $\ldots$).
- Understanding connectives in terms of subtyping is out of reach of simple programmers

Give a set-theoretic semantics to types

Types as sets of values and semantic subtyping

$\mathtt{T} ::= \mathtt{Bool} \mid \mathtt{Int} \mid \mathtt{Any} \mid \mathtt{(T\,,T)} \mid \mathtt{T} \vee \mathtt{T} \mid \mathtt{T} \,\&\, \mathtt{T} \mid \mathtt{not(T)} \mid \mathtt{T\text{-->}T}$

Each type *denotes* a set of values:

$\mathtt{Bool}$ is the set that contains just two values $\{\mathtt{true}\,,\mathtt{false}\}$

$\mathtt{Int}$ is the set of all the numeric constants: $\{\mathtt{0}, \mathtt{-1}, \mathtt{1}, \mathtt{-2}, \mathtt{2}, \mathtt{-3}, \ldots\}$.

$\mathtt{Any}$ is the set of *all* values.

$(\mathtt{T}_1\,,\mathtt{T}_2)$ is the set of all the pairs (v_1, v_2) where v_1 is a value in $\mathtt{T}_1$ and v_2 a value in $\mathtt{T}_2$, that is $\{(v_1, v_2) \mid v_1 \in \mathtt{T}_1\,,\, v_2 \in \mathtt{T}_2\}$.

$\mathtt{T}_1 \vee \mathtt{T}_2$ is the *union* of the sets $\mathtt{T}_1$ and $\mathtt{T}_2$, that is $\{v \mid v \in \mathtt{T}_1 \text{ or } v \in \mathtt{T}_2\}$

$\mathtt{T}_1 \,\&\, \mathtt{T}_2$ is the *intersection* of the sets $\mathtt{T}_1$ and $\mathtt{T}_2$, i.e. $\{v \mid v \in \mathtt{T}_1 \text{ and } v \in \mathtt{T}_2\}$.

$\mathtt{not(T)}$ is the set of all the values not in $\mathtt{T}$, that is $\{v \mid v \notin \mathtt{T}\}$.
In particular $\mathtt{not(Any)}$ is the empty set (written $\mathtt{Empty}$).

$\mathtt{T}_1\mathtt{\text{-->}}\mathtt{T}_2$ is the set of all function values that when applied to a value in $\mathtt{T}_1$, if they return a value, then this value is in $\mathtt{T}_2$.

Semantic subtyping

Subtyping is set-containment

Outline

Set-theoretic types in Perl 6

A function *value* is a λ-abstraction. In Perl6 it is any expression of the form:

```
    sub (parameters){body}
```

For instance (functions can be named):

```
    sub succ(Int $x){ $x + 1 }
```

the `succ` function is a value in/of type `Int-->Int`.

Subtypes can be defined intensionally:

```
    subset Even of Int where { $_ % 2 == 0 }
    subset Odd  of Int where { $_ % 2 == 1 }
```

Clearly:

both `succ:Even-->Odd` and `succ:Odd-->Even`

therefore:

```
    succ :  (Even-->Odd) & (Odd-->Even)
```

Subtyping

Notice that every function value in `(Even-->Odd) & (Odd-->Even)` is also in `Int-->Int`. Thus:

```
(Even-->Odd) & (Odd-->Even) <: Int-->Int
```

The converse does not hold: identity `sub(Int $x){ $x }` is a counterexample.

The above is just an instance of the following relation

$$(\mathtt{S}_1\texttt{-->}\mathtt{T}_1)\ \&\ (\mathtt{S}_2\texttt{-->}\mathtt{T}_2)\ \texttt{<:}\ (\mathtt{S}_1 \vee \mathtt{S}_2)\texttt{-->}(\mathtt{T}_1 \vee \mathtt{T}_2) \tag{4}$$

that holds for all types, $\mathtt{S}_1$, $\mathtt{S}_2$, $\mathtt{T}_1$, and $\mathtt{T}_2$,

The relation (4) shows why defining subtyping for type connectives is far more difficult than just with constructors: connectives *mix* types of different forms.

Overloaded functions

Overloaded functions are defined by giving multiple definitions of the same function prefixed by the `multi` modifier:

```
multi sub sum(Int $x, Int $y) { $x + $y }
multi sub sum(Bool $x, Bool $y) { $x && $y }
```

$$\texttt{sum} : (\texttt{(Int,Int)-->Int}) \;\&\; (\texttt{(Bool,Bool)-->Bool}), \tag{5}$$

Just one parameter is enough for selection. The *curried* form is equivalent.

```
multi sub sumC(Int  $x){ sub (Int  $y){$x + $y } }
multi sub sumC(Bool $x){ sub (Bool $y){$x && $y} }
```

In Perl we can use "`;;`" to separate parameters used for code selection from those passed to the selected code:

```
multi sub sumC(Int $x ;; Int $y) { $x + $y }
multi sub sumC(Bool $x ;; Bool $y) { $x && $y }
```

Both definitions of `sumC` have type

$$(\texttt{Int-->}(\texttt{Int-->Int})) \;\&\; (\texttt{Bool-->}(\texttt{Bool-->Bool})). \tag{6}$$

though partial application is possible only with the first definition of `sumC`

Dynamic dispatch

Dynamic dispatch

The code to execute for a multisubroutine is chosen at run-time according to the type of the argument.
The multi-subroutine with the *best* approximating input type is executed

- All examples given so far can be resolved at static time
- Dynamic dispatch is sensible only when types change during computation.

In a statically-typed language with subtyping, the type of an expression may decrease during the computation.

Example:

```
( sub(Int $x){ $x % 4 } )(3+2)
```

`Int` at compile time; `Even` after the reduction.

Dynamic dispatch

Example

```
multi sub mod2sum(Even $x , Odd $y) { 1 }
multi sub mod2sum(Odd $x , Even $y) { 1 }
multi sub mod2sum(Int $x , Int $y) { 0 }
```

Its type (with singleton types: *v* is the type that contains just value *v*)

$$
\begin{array}{rl}
 & \big(\texttt{(Even , Odd)-->1}\big) \\
\& & \big(\texttt{(Odd , Even)-->1}\big) \\
\& & \big(\texttt{(Int,Int)-->0} \vee \texttt{1}\big)
\end{array}
$$

Exercise

Find a more precise type and justify how the type checker can deduce it.

Formation rules for multi-subroutines: Ambigous Selection

Alternative definition for `mod2sum`:

```
multi sub mod2sum(Even $x , Int $y){ $y % 2 }
multi sub mod2sum(Int $x , Odd $y){ ($x+1) % 2 }
```

Mathematically correct but selection is ambigous: the computation is stuck on arguments of type `(Even,Odd)`.

Formation rule 1: Ambiguity

A multi-subroutine is *free from ambiguity* if whenever it has definitions for input `S` and `T`, and `S & T` is not empty, then it has a definition for input `S & T`.

It is a *formation rule*. It belongs to language design not to the type system:

$$\texttt{((Even,Int) --> 0}\vee\texttt{1) \& ((Int,Odd) --> 0}\vee\texttt{1)}$$

the type above is perfectly ok (and a correct type for `mod2sum`).

Formation rules for multi-subroutines: Specialization

Because of dynamic dispatch during the execution:

- the type of the argument changes, $\Rightarrow$
- the code selected for a multi-subroutine changes, $\Rightarrow$
- the type of application changes

Types may *only* decrease along the computation

Consider again:

```
multi sub mod2sum(Even $x , Odd $y) { 1 }
multi sub mod2sum(Odd $x , Even $y) { 1 }
multi sub mod2sum(Int $x , Int $y) { 0 }
```

which has type

$$\texttt{((Even,Odd)-->1) \& ((Odd,Even)-->1) \& ((Int,Int)-->0} \vee \texttt{1)}$$

For the application `mod2sum(3+3,3+2)`:

- static time: third code selected; static type is $0 \vee 1$
- run time: first code selected; dynamic type is 1 (notice 1 `<:` $0 \vee 1$)

Formation rules for multi-subroutines: Specialization

"Types may *only* decrease along the computation"

Why does it matter?

```
multi sub foo(Int $x) { $x+42 }
multi sub foo(Odd $x) { true }
```

Consider `10+(foo(3+2))`: statically well-typed but yields a runtime type error.

How to ensure it for dynamic dispatch?

Formation rule 2: Specialization

A multi-subroutine is *specialization sound* if whenever it has definitions for input S and T, and S<:T, then the definition for input S returns a type smaller than the one returned by the definition for T.

Example:

```
multi sub foo(S₁ $x) returns T₁ { ... }
multi sub foo(S₂ $x) returns T₂ { ... }
```

Specialization sound: If S_1<:S_2 then T_1<:T_2.

Formation rules for multi-subroutines: Specialization

Once more, a *formation rule*: concerns language design, not the type system. The type system is perfectly happy with the type

$$(\mathtt{S}_1\texttt{-->}\mathtt{T}_1)\ \&\ (\mathtt{S}_2\texttt{-->}\mathtt{T}_2)$$

even if $\mathtt{S}_1\texttt{<:}\mathtt{S}_2$ and $\mathtt{T}_1$ and $\mathtt{T}_2$ are not related. However consider all the possible cases of applications of a function of this type:

1. If the argument is in $\mathtt{S}_1\ \&\ \mathtt{S}_2$, then the application has type $\mathtt{T}_1\ \&\ \mathtt{T}_2$.
2. If the argument is in $\mathtt{S}_1 \backslash \mathtt{S}_2$ and case 1 does not apply, then the application has type $\mathtt{T}_1$.
3. If the argument is in $\mathtt{S}_2 \backslash \mathtt{S}_1$ and case 1 does not apply, then the application has type $\mathtt{T}_2$.
4. If the argument is in $\mathtt{S}_1 \vee \mathtt{S}_2$ and no previous case applies, then the application has type $\mathtt{T}_1 \vee \mathtt{T}_2$.

Formation rules for multi-subroutines: Specialization

This case

1. If the argument is in $\texttt{S}_1 \;\&\; \texttt{S}_2$, then the application has type $\texttt{T}_1 \;\&\; \texttt{T}_2$.

may confuse the programmer when $\texttt{S}_2\texttt{<:}\texttt{S}_1$, since in this case $\texttt{S}_2 = \texttt{S}_2 \;\&\; \texttt{S}_1$:

When a function of type $(\texttt{S}_1\texttt{-->}\texttt{T}_1) \;\&\; (\texttt{S}_2\texttt{-->}\texttt{T}_2)$ with $\texttt{S}_2\texttt{<:}\texttt{S}_1$, is applied to an argument of type $\texttt{S}_2$, then the application returns results in $\texttt{T}_1 \;\&\; \texttt{T}_2$.

Design choice: to avoid confusion force (wlog) the programmer to specify that the return type for a $\texttt{S}_2$ input is (some subtype of) $\texttt{T}_1 \;\&\; \texttt{T}_2$.

This can be obtained by accepting only specialization sound definitions and greatly simplifies the presentation of the type discipline of the language.

Outline

Covariance and contravariance

Homework assignment:

1. Mandatory: Study the covariance and contravariance problem described in the first 3 sections of the following paper (click on the title).

   G. Castagna. Covariance and Contravariance: a fresh look at an old issue. *Draft manuscript, 2014.*

2. Optional: if you want to know what is under the hood, you can read Section 4 of the same paper, which describes a state-of-the-art implementation of a type system with set-theoretic types.

Outline

CDuce is built on types

The main motivation for studying set-theoretic types is to define strongly typed programming languages for XML.

CDuce is a programming language for XML whose design is completely based on set-theoretic types.

In CDuce set-theoretic types are pervasive:

1. XML types are encoded in set-theoretic types
2. Patterns are types with capture variables
3. Set-theoretic types are used for informative error messages
4. Types are used for efficient JIT compilation

XML syntax

```
type Bib = <bib>[Book*]             Kleene star
type Book = <book year=String>[                 attribute types
                     Title                      nested elements
                     (Author+ | Editor+)                 unions
                     Price?                      optional elems
                     PCDATA]                      mixed content
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]                             PCDATA
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
```

This and: singletons, intersections, differences, `Empty`**, and** `Any`**.**

We saw that all this can be encoded with recursive and set-theoretic types

Types & patterns: the functional languages perspective

- **Types** are sets of **values**
- Values are decomposed by **patterns**
- Patterns are roughly values with **capture variables**

Instead of

```
let x = fst(e) in
let y = snd(e) in (y,x)
```

with patterns one can write

```
let (x,y) = e in (y,x)
```

which is syntactic sugar for

```
match e with (x,y) -> (y,x)
```

"`match`" is more interesting than "`let`", since it can test several "**|**"-separated patterns.

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
  match x with
   | (‘nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards, constants.

But if we:

1. **use for types the same constructors as for values**
   (e.g. (s,t) **instead of** $s \times t$**)**
2. **use values to denote singleton types**
   (e.g. `‘nil` **in the list type);**
3. **consider the wildcard “_” as synonym of** `Any`

Patterns in CDuce

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
```

PATTERNS

```
<bib>[(x::<book year="1990">[ _* Publisher\"ACM"] | _)*]
```

Returns all the captured books

Exact type inference:

E.g.: if we match the pattern `[(x::Int|_)**]` against an expression of type `[Int* String Int]` the type deduced for x is `[Int+]`

Outline

Functions in CDuce

Functions: basic usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Extract subsequences (union polymorphism)

```
fun (Invited|Talk -> [Author+])
    <_>[ Title x::Author* ] -> x
```

Extract subsequences of non-consecutive elements:

```
fun ([(Invited|Talk|Event)*] -> ([Invited*], [Talk*]))
    [ (i::Invited | t::Talk | _)* ] -> (i,t)
```

Perl-like string processing (`String` = `[Char*]`)

```
fun parse_email (String -> (String,String))
    | [ local::_* '@' domain::_* ] -> (local,domain)
    | _ -> raise "Invalid email address"
```

Functions: advanced usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Functions can be **higher-order** and **overloaded**

```
let patch_program
(p :[Program], f :(Invited -> Invited) & (Talk -> Talk)):[Program]
   = xtransform p with (Invited | Talk) & x -> [ (f x) ]
```

Higher-order, overloading, subtyping provide name/code sharing...

```
let first_author ([Program] -> [Program];
                  Invited -> Invited;
                  Talk -> Talk)
| [ Program ] & p -> patch_program (p,first_author)
| <invited>[ t a _* ] -> <invited>[ t a ]
| <talk>[ t a _* ] -> <talk>[ t a ]
```

Even more compact: replace the last two branches with:

```
<(k)>[ t a _* ] -> <(k)>[ t a ]
```

Red-black trees in CDuce

```
type RBtree = Btree | Rtree;;
type Btree  = <black elem=Int>[ RBtree RBtree ]  | [] ;;
type Rtree  = <red   elem=Int>[ Btree Btree ];;

type Wrongtree  = Wrongleft | Wrongright;;
type Wrongleft  = <red elem=Int>[ Rtree Btree ];;
type Wrongright = <red elem=Int>[ Btree Rtree ];;
type Unbalanced = <black elem=Int>([Wrongtree RBtree] | [RBtree Wrongtree])

let balance ( Unbalanced -> Rtree ; Rtree -> Rtree ; Btree\[] -> Btree\[] ;
              [] -> [] ; Wrongleft -> Wrongleft ; Wrongright -> Wrongright)
  | <black (z)>[ <red (y)>[ <red (x)>[ a b ] c ] d ]
  | <black (z)>[ <red (x)>[ a <red (y)>[ b c ] ] d ]
  | <black (x)>[ a <red (z)>[ <red (y)>[ b c ] d ] ]
  | <black (x)>[ a <red (y)>[ b <red (z)>[ c d ] ] ] ->
        <red (y)>[  <black (x)>[ a b ]   <black (z)>[ c d ]  ]
  | x -> x

let insert (x : Int) (t : Btree) : Btree =
 let ins_aux ( [] -> Rtree ; Btree\[] -> RBtree\[]; Rtree -> Rtree|Wrongtree)
    | [] -> <red elem=x>[ [] [] ]
    | (<(color) elem=y>[ a b ]) & z ->
           if x << y then balance <(color) elem=y>[ (ins_aux a) b ]
           else if x >> y then balance <(color) elem=y>[ a (ins_aux b) ]
           else z
   in match ins_aux t with
      | <_ (y)>[ a b ] -> <black (y)>[ a b ]
```

Red-black trees in Polymorphic CDuce

```
type RBtree = Btree | Rtree;;
type Btree  = <black elem=Int>[ RBtree RBtree ]  | [] ;;
type Rtree  = <red   elem=Int>[ Btree Btree ];;

type Wrongtree  = <red elem=Int>[ Rtree Btree ]
                | <red elem=Int>[ Btree Rtree ];;
type Unbalanced = <black elem=Int>([Wrongtree RBtree] | [RBtree Wrongtree])

let balance ( Unbalanced -> Rtree ; α\Unbalanced -> α\Unbalanced )
  | <black (z)>[ <red (y)>[ <red (x)>[ a b ] c ] d ]
  | <black (z)>[ <red (x)>[ a <red (y)>[ b c ] ] d ]
  | <black (x)>[ a <red (z)>[ <red (y)>[ b c ] d ] ]
  | <black (x)>[ a <red (y)>[ b <red (z)>[ c d ] ] ] ->
        <red (y)>[  <black (x)>[ a b ]   <black (z)>[ c d ]  ]
  | x -> x

let insert (x : Int) (t : Btree) : Btree =
 let ins_aux ( [] -> Rtree ; Btree\[] -> RBtree\[]; Rtree -> Rtree|Wrongtree)
    | [] -> <red elem=x>[ [] [] ]
    | (<(color) elem=y>[ a b ]) & z ->
           if x << y then balance <(color) elem=y>[ (ins_aux a) b ]
           else if x >> y then balance <(color) elem=y>[ a (ins_aux b) ]
           else z
   in match ins_aux t with
       | <_ (y)>[ a b ] -> <black (y)>[ a b ]
```

Outline

Informative error messages

List of books of a given year, stripped of the Editors and Price

```
fun onlyAuthors (year:Int,books:[Book*]):[Book*] =
 select <book year=y>(t@a) from
   <book year=y>[ t::Title   a::Author+  _* ] in books
 where int_of(y) = year
```

Returns the following error message:

```
Error at chars 81-83:
   select <book year=y>(t@a) from
This expression should have type:
[ Title (Editor+|Author+) Price?  ]
but its inferred type is:
[ Title Author+ | Title ]

[ <title>[ ] ]
```

Efficient execution

Idea: if types tell you that something cannot happen, don't test it.

```
type A = <a>[A*]
type B = <b>[B*]

fun check(x : A|B) = match x with   A  -> 1 | B -> 0
fun check(x : A|B) = match x with <a>_ -> 1 | _  -> 0
```

- No backtracking.
- Whole parts of the matched data are not checked

Computing the optimal solution requires to fully exploit intersections and differences of types

Outline

Toolkit

Every programming language needs tools / libraries / DLS extensions.

Available for CDuce:

- OCaml full integration
- Web-services API
- Navigational patterns (à la XPath) [experimental]

CDuce↔OCaml Integration

A CDuce application that requires OCaml code

- Reuse existing librairies
  - Abstract data structures : hash tables, sets, ...
  - Numerical computations, system calls
  - Bindings to C libraries : databases, networks, ...
- Implement complex algorithms

An OCaml application that requires CDuce code

- CDuce used as an XML input/output/transformation layer
  - Configuration files
  - XML serialization of datas
  - XHTML code production

Need to seamlessly call OCaml code in CDuce and viceversa

Main Challenges

1. **Seamless integration:**
   No explicit conversion function in programs:
   the compiler performs the conversions
2. **Type safety:**
   No explicit type cast in programs:
   the standard type-checkers ensure type safety

What we need:

A mapping between OCaml and CDuce *types* and *values*

How to integrate the two type systems?

The translation can go just one way: OCaml → CDuce

⊕ **CDuce uses (semantic) subtyping; OCaml does not**
If we translate CDuce types into OCaml ones :
- soundness requires the translation to be monotone;
- no subtyping in Ocaml implies a constant translation;
⇒ *CDuce typing would be lost.*

⊕ **CDuce has unions, intersections, differences, heterogeneous lists; OCaml does not**
⇒ *OCaml types are not enough to translate CDuce types.*

⊖ **OCaml supports type polymorphism; CDuce does not yet (it does in the development version).**
⇒ *Polymorphic OCaml libraries/functions must be first instantied to be used in CDuce*

In practice

1. Define a mapping $\mathbb{T}$ from OCaml types to CDuce types.

t (*OCaml*)	$\mathbb{T}(t)$ (*CDuce*)
`int`	`min_int--max_int`
`string`	`Latin1`
$t_1 * t_2$	$(\mathbb{T}(t_1), \mathbb{T}(t_2))$
$t_1 \rightarrow t_2$	$\mathbb{T}(t_1) \rightarrow \mathbb{T}(t_2)$
t `list`	$[\mathbb{T}(t)*]$
t `array`	$[\mathbb{T}(t)*]$
t `option`	$[\mathbb{T}(t)?]$
t `ref`	`ref` $\mathbb{T}(t)$
A_1 of $t_1 \mid \ldots \mid A_n$ of t_n	$(`A_1, \mathbb{T}(t_1)) \mid \ldots \mid (`A_n, \mathbb{T}(t_n))$
$\{l_1 = t_1; \ldots; l_n = t_n\}$	$\{l_1 = \mathbb{T}(t_1); \ldots; l_n = \mathbb{T}(t_n)\}$

2. Define a retraction pair between OCaml and CDuce values.

`ocaml2cduce`: $t \rightarrow \mathbb{T}(t)$

`cduce2ocaml`: $\mathbb{T}(t) \rightarrow t$

Calling OCaml from CDuce

Easy

Use `M.f` to call the function `f` exported by the OCaml module `M`

The CDuce compiler checks type soundness and then
- applies `cduce2ocaml` to the arguments of the call
- calls the OCaml function
- applies `ocaml2cduce` to the result of the call

Example: use ocaml-mysql library in CDuce

```
let db = Mysql.connect Mysql.defaults;;

match Mysql.list_dbs db ‘None [] with
| (‘Some,l) -> print [ 'Databases: ' !(string_of l) '\ n' ]
| ‘None -> [];;
```

Calling CDuce from OCaml

Needs little work

Compile a CDuce module as an OCaml binary module by providing a OCaml (`.mli`) interface. Use it as a standard Ocaml module.

The CDuce compiler:

1. Checks that if `val` f:t in the `.mli` file, then the CDuce type of f is a *subtype* of $\mathbb{T}(t)$
2. Produces the OCaml glue code to export CDuce values as OCaml ones and bind OCaml values in the CDuce module.

Example: use CDuce to compute a factorial:

```
(* File cdnum.mli: *)
val fact: Big_int.big_int -> Big_int.big_int

(* File cdnum.cd: *)
let aux ((Int,Int) -> Int)
| (x, 0 | 1) -> x
| (x, n) -> aux (x * n, n - 1)

let fact (x : Int) : Int = aux(1,x)
```

Concurrency

Outline

Outline

Concurrent vs parallel

- **Concurrency**
  - Do many unrelated things “at once”
  - Goals are expressiveness, responsiveness, and multitasking
- **Parallelism**
  - Get a faster answer with multiple CPUs

Here we will focus on the concurrency part

Threads

Thread

Threads are sequential computations that share memory.

Threads of control (aka lightweight process) execute concurrently in the same memory space. Threads communicate by in-place modifications of shared data structures, or by sending and receiving data on communication channels.

Two kinds of threads

1. *Native threads (a.k.a. OS Threads)*. They are directly handled by the OS.
    - Compatible with multiprocessors and low level processor capabilities
    - Better handling of input/output.
    - Compatible with native code.
2. *Green threads (a.k.a. light threads or user space threads)*. They are handled by the virtual machine.
    - More lightweight: context switch is much faster, much more threads can coexist.
    - They are portable but must be executed in the VM.
    - Input/outputs must be asynchronous since a blocking system call blocks all the threads within the process.

Which threads for whom

Green threads To be used if:

- Don’t want to wait for user input or blocking operations
- Need a lot of threads and need to switch from one to another rapidly
- Don’t care about about using multiple CPUs, since "the machine spends most of its time waiting on the user anyway".
- Typical usage: a web server.

Native threads To be used if:

- Don’t want to wait for long running computations
- Either long running computation must advance “at the same time” or, better, run in parallel on multiple processors and actually finish faster
- Typical usage: heavy computations

Haskell mixed solution

Native threads: *1:1* Threre is a one-to-one correspondence between the application-level threads and the kernel threads

Green threads: *N:1* the program threads are managed in the user space. The kernel is not aware of them, so all application-level threads are mapped to a single kernel thread.

Haskell and Erlang solution:

Hybrid threads: *N:M* (with $N \geqslant M$) Intermediate solution: spawn a whole bunch of lightweight green threads, but the interpreter schedules these threads onto a smaller number of native threads.

- ⊕ Can exploit multi-core, multi-processor architectures
- ⊕ Avoids to block all the threads on a blocking call
- ⊖ Hard to implement in particular the scheduling.
- ⊖ When using blocking system calls you actually need to notify somehow kernel to block only one green thread and not kernel one.

Multi-threading

Two kinds of multi-threading

1. *Preemptive threading:* A scheduler handles thread executions. Each thread is given a maximum time quantum and it is interrupted either because it finished its time slice or because it requests a “slow” operation (e.g., I/O, page-faulting memory access ...)
2. *Cooperative threading:* Each thread keeps control until either it explicitly handles it to another thread or it execute an asynchronous operation (e.g. I/O).

Possible combinations

1. Green threads are mostly preemptive, but several implementations of cooperative green threads are available (eg, the `Lwt` library in OCaml and the Coro module in Perl).
2. OS threads are nearly always preemptive since on a cooperative OS all applications must be programmed “fairly” and pass the hand to other applications from time to time

Outline

Shared memory model/ Process synchronization

Threads/processes are defined to achieve together a common goal therefore they do not live in isolation:

- To ensure that the goal is achieved threads/processes must *synchronize*.
- The purpose of *process synchronization* is to enforce constraints such as:
    - Serialization: this part of thread A must happen before this part of thread B.
    - Mutual exclusion: no two threads can execute this concurrently.

Several software tools are available to build synchronization policies for shared memory accesses:

- Semaphores
- Locks / Mutexes / Spinlocks
- Condition variables
- Barriers
- Monitors

Concurrent events

Two events are concurrent if we cannot tell by looking at the program which will happen first.

Thread A

```
a1 x = 5
a2 print x
```

Thread B

```
b1 x = 7
```

Possible outcomes:

- output 5 and final value for `x` = 7 (eg, a1→a2→b1)
- output 7 and final value for `x` = 7 (eg, a1→b1→a2)
- output 5 and final value for `x` = 5 (eg, b1→a1→a2)

Thread A

```
x = x + 1
```

Thread A

```
x = x + 1
```

If initially `x = 0` then both `x = 1` and `x = 2` are possible outcomes

Reason: The increment may be **not atomic**: $(t \leftarrow \texttt{read}\ x; x \leftarrow \texttt{read}\ t)$

For instance, in some assembler, `LDA $44; ADC #$01; STA $44` instead of `INC $44`

Model of execution

We must define the model of execution

- On some machines `x++` is atomic
- But let us not count on it: we do not want to write specialized code for each different hardware.
- Assume (rather pessimistically) that:
  - Result of concurrent writes is undefined.
  - Result of concurrent read-write is undefined.
  - Concurrent reads are ok.
  - Threads can be interrupted at any time (preemptive multi-threading).

To solve synchronization problems let us first consider a very simple and universal software synchronization tool: *semaphores*

Semaphore

Semaphores are

- ⊕ *Simple*. The concept is just a little bit harder than that of a variable.
- ⊕ *Versatile*. You can pretty much solve all synchronization problems by semaphores.
- ⊖ *Error-prone.* They are so low level that they tend to be error-prone.

We start by them because:

- They are good for learning to think about synchronization

However:

- They are *not* the best choice for common use-cases (you'd better use specialized tools for specific problems, such as mutexes, conditionals, monitors, etc).

Definition (Dijkstra 1965)

A semaphore is an integer `s`$\geqslant 0$ with two operations P and S:
- $P(s)$: if `s>0` then `s--` else the caller is suspended
- $S(s)$: if there is a suspended process, then resume it else `s++`

Semaphore object

In Python:

A semaphore is a class encapsulating an integer with two methods:

- `Semaphore(`*n*`)` initialize the counter to *n* (default is 1).
- `acquire()` if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called release(). The order in which blocked threads are awakened is not specified.
- `release()` If another thread is waiting for it to become larger than zero again, wake up that thread otherwise increment the internal counter

Variations that can be met in other languages:

- `wait()`, `signal()` (I will use this pair, because of the *signaling* pattern).
- negative counter to count the process awaiting at the semaphore.

Notice: no `get` method (to return the value of the counter). **Why?**

Semaphores to enforce Serialization

Problem:

Thread A

```
statement a1
```

Thread B

```
statement b1
```

How do we *enforce* the constraint: « a1 before b1 » ?

The signaling pattern:

```
sem = Semaphore(0)
```

Thread A

```
statement a1
sem.signal()
```

Thread B

```
sem.wait()
statement b1
```

You can think of `Semaphore(0)` as a locked lock.

Semaphores to enforce Mutual Exclusion

Problem:

Thread A

```
x = x + 1
```

Thread B

```
x = x + 1
```

Concurrent execution is non-deterministic

How can we *avoid* concurrent access?

Solution:

```
mutex = Semaphore(1)
```

Thread A

```
mutex.wait()
  x = x + 1
mutex.signal()
```

Thread B

```
mutex.wait()
  x = x + 1
mutex.signal()
```

Code between `wait` and `signal` is atomic.

More synch problems: readers and writers

Problem:

Threads are either writers or readers:

- Only *one* writer can *write* concurrently
- A reader cannot *read* concurrently with a writer
- Any number of readers can *read* concurrently

Solution:

```
readers = 0
mutex = Semaphore(1)
roomEmpty = Semaphore(1)
```

Writer threads

```
roomEmpty.wait()
   critical section for writers
roomEmpty.signal()
```

Reader threads

```
mutex.wait()
  readers += 1
  if readers == 1:
     roomEmpty.wait()  # first in lock
mutex.signal()
     critical section for readers
mutex.wait()
  readers -= 1
  if readers == 0:
     roomEmpty.signal() # last out unlk
mutex.signal()
```

Readers and writers

Let us look for some common patterns

- The scoreboard pattern (readers)
  - Check in
  - Update state on the scoreboard (number of readers)
  - make some conditional behavior
  - check out
- The turnstile pattern (writer)
  - Threads go through the turnstile serially
  - One blocks, all wait
  - It passes, it unblocks
  - Other threads (ie, the readers) can lock the turnstile

Readers and writers

Readers while checking in/out implement the *lightswitch* pattern:

- The first person that enters the room switch the light on (acquires the lock)
- The last person that exits the room switch the light off (releases the lock)

Implementation:

```
class Lightswitch:
  def __init__(self):
      self.counter = 0
      self.mutex = Semaphore(1)

  def lock(self, semaphore):
      self.mutex.wait()
          self.counter += 1
          if self.counter == 1:
              semaphore.wait()
      self.mutex.signal()

  def unlock(self, semaphore):
      self.mutex.wait()
          self.counter -= 1
          if self.counter == 0:
              semaphore.signal()
      self.mutex.signal()
```

Before:

```
readers = 0
mutex = Semaphore(1)
roomEmpty = Semaphore(1)
```

Writer threads

```
roomEmpty.wait()
   critical section for writers
roomEmpty.signal()
```

Reader threads

```
mutex.wait()
  readers += 1
  if readers == 1:
     roomEmpty.wait()  # first in lock
mutex.signal()
     critical section for readers
mutex.wait()
  readers -= 1
  if readers == 0:
     roomEmpty.signal() # last out unlk
mutex.signal()
```

After:

```
readLightswitch = Lightswitch()
roomEmpty = Semaphore(1)
```

Writer threads

```
roomEmpty.wait()
   critical section for writers
roomEmpty.signal()
```

Reader threads

```
readLightswitch.lock(roomEmpty)
     critical section for readers
readLightswitch.unlock(roomEmpty)
```

Programming golden rules

When programming becomes too complex then:

1. Abstract common patterns
2. Split it in more elementary problems

The previous case was an example of abstraction. Next we are going to see an example of modularization, where we combine our elementary patterns to solve more complex problems

The unisex bathroom problem

A women at Xerox was working in a cubicle in the basement, and the nearest women's bathroom was two floors up. She proposed to the Uberboss that they convert the men's bathroom on her floor to a unisex bathroom.
The Uberboss agreed, provided that the following synchronization constraints can be maintained:

1. There cannot be men and women in the bathroom at the same time.
2. There should never be more than three employees squandering company time in the bathroom.

You may assume that the bathroom is equipped with all the semaphores you need.

The unisex bathroom problem

Solution hint:

```
    empty = Semaphore(1)
    maleSwitch = Lightswitch()
    femaleSwitch = Lightswitch()
    maleMultiplex = Semaphore(3)
    femaleMultiplex = Semaphore(3)
```

- `empty` is 1 if the room is empty and 0 otherwise.
- `maleSwitch` allows men to bar women from the room. When the first male enters, the lightswitch locks `empty`, barring women; When the last male exits, it unlocks `empty`, allowing women to enter. Women do likewise using `femaleSwitch`.
- `maleMultiplex` and `femaleMultiplex` ensure that there are no more than three men and three women in the system at a time (they are semaphores used as locks).

The unisex bathroom problem

A solution:

Female Threads

```
femaleSwitch.lock(empty)
    femaleMultiplex.wait()
        bathroom code here
    femaleMultiplex.signal()
femaleSwitch.unlock(empty)
```

Male Threads

```
maleSwitch.lock(empty)
    maleMultiplex.wait()
        bathroom code here
    maleMultiplex.signal()
maleSwitch.unlock(empty)
```

Any problem with this solution?

This solution allows starvation. A long line of women can arrive and enter while there is a man waiting, and vice versa.

Find a solution

Hint: Use a turnstile to access to the lightswitches: when a man arrives and the bathroom is already occupied by women, block turnstile so that more women cannot check the light and enter.

The no-starve unisex bathroom problem

```
   **turnstile = Semaphore(1)**
   empty = Semaphore(1)
   maleSwitch = Lightswitch()
   femaleSwitch = Lightswitch()
   maleMultiplex = Semaphore(3)
   femaleMultiplex = Semaphore(3)
```

Female Threads

```
turnstile.wait()
    femaleSwitch.lock(empty)
turnstile.signal()

    femaleMultiplex.wait()
        bathroom code here
    femaleMultiplex.signal()

femaleSwitch.unlock (empty)
```

Male Threads

```
turnstile.wait()
    maleSwitch.lock(empty)
turnstile.signal()

    maleMultiplex.wait()
        bathroom code here
    maleMultiplex.signal()

maleSwitch.unlock (empty)
```

Actually we could have used the same multiplex for both females and males.

Summary so far

- Solution composed of *patterns*
- Patterns can be encapsulated as objects or modules
- Unisex bathroom problem is a good example of use of both *abstraction* and *modularity* (lightswitches and turnstiles)

- Unfortunately, patterns *often* interact and interfere. Hard to be confident of solutions (formal verification and test are not production-ready yet).
- Especially true for semaphores which are very low level:
  - ⊕ They can be used to implement more complex synchronization patterns.
  - ⊖ This makes interference much more likely.

Before discussing more general problems of shared memory synchronization, let us introduced some higher-level and more specialized tools that, being more specific, make interference less likely.

- Locks
- Conditional Variables
- Monitors

Outline

Locks

Locks are like those on a room door:

- *Lock acquisition:* A person enters the room and locks the door. Nobody else can enter.
- *Lock release:* The person in the room exits unlocking the door.

Persons are *threads*, rooms are *critical regions*.
A person that finds a door locked can either wait or come later (somebody lets it know that the room is available).

Similarly there are two possibilities for a thread that failed to acquire a lock:

1. It keeps trying. This kind of lock is a *spinlock*. Meaningful only on multi-processors, they are common in High performance computing (where most of the time each thread is scheduled on its own processor anyway).
2. It is suspended until somebody signals it that the lock is available. The only meaningful lock for uniprocessor. This kind of lock is also called *mutex* (but often *mutex* is used as a synonym for lock).

Difference between a mutex and a binary semaphore

A mutex is *different* from a binary semaphore (ie, a semaphore initialized to 1), since it combines the notion of *exclusivity of manipulation* (as for semaphores) with others extra features such as *exclusivity of possession* (only the process which has taken a mutex can free it) or *priority inversion protection*. The differences between mutexes and semaphores are operating system/language dependent, though mutexes are implemented by specialized, faster routines.

Example

What follows can be done with semaphore `s` but not with a mutex, since B unlocks a lock of A (cf. the signaling pattern):

```
Thread A                     Thread B
   ⋮                            ⋮
some stuff                   some stuff
   ⋮                            ⋮
wait(s)                      signal(s) (* A can continue *)
   ⋮                            ⋮
some other stuff
```

Mutex

Since semaphores are *for what concerns mutual exclusion* a simplified version of mutexes, it is clear that mutexes have operations very similar to the former:

- A `init` or `create` operation.
- A `wait` or `lock` operation that tries to acquire the lock and suspends the thread if it is not available
- A `signal` or `unlock` operation that releases the lock and possibly awakes a thread waiting for the lock
- Sometimes a `trylock`, that is, a non blocking locking operation that returns an error or false if the lock is not available.

A mutex is *reentrant* if the same thread can acquire the lock multiple times. However, the lock must be released the same number of times or else other threads will be unable to acquire the lock.

Nota Bene: A reentrant mutex has some similarities to a counting semaphore: the number of lock acquisitions is the counter, but only one thread can successfully perform multiple locks (exclusivity of possession).

Implementation of locks

Some examples of lock implementations:

- Using hardware special instructions like `test-and-set` or `compare-and-swap`
- Peterson algorithm (spinlock, deadlock free) in Python:

```
flag=[0,0]
turn = 0        # initially the priority is for thread 0
```

Thread 0

```
flag[0] = 1
turn = 1
while flag[1] and turn : pass
    critical section
flag[0] = 0
```

Thread 1

```
flag[1] = 1
turn = 0
while flag[0] and not turn : pass
    critical section
flag[1] = 0
```

`flag[i] == 1`: Thread `i` wants to enter;

`turn == i` it is the turn of Thread `i` to enter, if it wishes.

- Lamport's bakery algorithm (deadlock and starvation free)

  Every threads modifies only its own variables and accesses to other variables only by reading.

Condition Variables

Locks provides a passive form of synchronization: they allow waiting for shared data to be free, but do not allow waiting for the data to have a particular state. *Condition variables* are the solution to this problem.

Definition

A *condition variable* is an atomic waiting and signaling mechanism which allows a process or thread to atomically stop execution and release a lock until a signal is received.

Rationale

It allows a thread to sleep inside a critical region without risk of deadlock.

Three main operations:

- `wait()` releases the lock, gives up the CPU until signaled and then re-acquire the lock.
- `signal()` wakes up a thread waiting on the condition variable, if any.
- `broadcast()` wakes up all threads waiting on the condition.

Condition Variables

Note: The term "condition variable" is misleading: it does not rely on a variable but rather on signaling at the system level. The term comes from the fact that condition variables are most often used to notify changes in the state of shared variables, such as in

- Notify a reader thread that a writer thread has filled its data set.
- Notify consumer processes that a producer thread has updated a shared data set.

Semaphores and Condition variables

- semaphores and condition variables both use `wait` and `signal` as valid operations,
- the purpose of both is somewhat similar, but
- *they are different:*
    - With a semaphore the signal operation increments the value of the semaphore even if there is no blocked process. The signal is remembered.
    - If there are no processes blocked on the condition variable then the signal function does nothing. The signal is not remembered.
    - With a semaphore you must be careful about deadlocks.

Monitors

Motivation:

- Semaphores are incredibly versatile.
- The problem with them is that they are *dual purpose*: they can be used for both *mutual exclusion* and *scheduling* constraints. This makes the code hard to read, and hard to get right.
- In the previous slides we have introduced two separate constructs for each purpose: mutexes and conditional variables.
- Monitors groups them together (keeping each disctinct from the other) to protect some shared data:

Definition (Monitor)

a lock and zero or more condition variables for managing concurrent access to shared data by defining some given operations.

Monitors

Example: a synchronized queue

In pseudo-code:

```
monitor SynchQueue {
  lock = Lock.create
  condition = Condition.create

  addToQueue(item) {
    lock.acquire();
    put item on queue;
    condition.signal();
    lock.release();
  }

  removeFromQueue() {
    lock.acquire();
    while nothing on queue do
       condition.wait(lock)    // release lock; go to
    done                       // sleep; re-acquire lock
    remove item from queue;
    lock.release();
    return item
  }
}
```

Different kinds of Monitors

Need to be careful about the precise definition of signal and wait:

Mesa-style: (Nachos, most real operating systems)

- Signaler keeps lock, processor
- Waiter simply put on ready queue, with no special priority. (in other words, waiter may have to wait for lock)

Hoare-style: (most textbooks)

- Signaler gives up lock, CPU to waiter; waiter runs immediately
- Waiter gives lock, processor back to signaler when it exits critical section or if it waits again.

Above code for synchronized queuing happens to work with either style, but for many programs it matters which one you are using. With Hoare-style, can change "while" in removeFromQueue to an "if", because the waiter only gets woken up if item is on the list. With Mesa-style monitors, waiter may need to wait again after being woken up, because some other thread may have acquired the lock, and removed the item, before the original waiting thread gets to the front of the ready queue.

Preemptive threads in OCaml

Four main modules:

- Module `Thread`: lightweight threads (abstract type `Thread.t`)
- Module `Mutex`: locks for mutual exclusion (abstract type `Mutex.t`)
- Module `Condition`: condition variables to synchronize between threads (abstract type `Condition.t`)
- Module `Event`: first-class synchronous channels (abstract types `'a Event.channel` and `'a Event.event`)

Two implementations:

- System threads. Uses OS-provided threads: POSIX threads for Unix, and Win32 threads for Windows. Supports both bytecode and native-code.
- Green threads. Time-sharing and context switching at the level of the bytecode interpreter. Works on OS without multi-threading but cannot be used with native-code programs.

Nota Bene: Always work on a single processor (because of OCaml's GC). No advantage from multi-processors (apart from explicit execution of C code or system calls): threads are just for structuring purposes.

Module `Thread`

- `create : ('a -> 'b) -> 'a -> Thread.t`
  `Thread.create f e` creates a new thread of control, in which the function application `f(e)` is executed concurrently with the other threads of the program.
- `kill : Thread.t -> unit`
  `kill p` terminates prematurely the thread `p`
- `join : Thread.t -> unit`
  `join p` suspends the execution of the calling thread until the termination of `p`
- `delay : float -> unit`
  `delay d` suspends the execution of the calling thread for `d` seconds.

```
# let f () = for i=0 to 10 do Printf.printf "(%d)" i done;;
val f : unit -> unit = <fun>
#  Printf.printf "begin ";
   Thread.join (Thread.create f ());
   Printf.printf " end";;
begin (0)(1)(2)(3)(4)(5)(6)(7)(8)(9)(10) end- : unit = ()
```

Module Mutex

- create : unit -> Mutex.t Return a new mutex.
- lock : Mutex.t -> unit Lock the given mutex.
- try_lock : Mutex.t -> bool Non blocking lock.
- unlock : Mutex.t -> unit Unlock the given mutex.

Dining philosophers

Five philosophers sitting at a table doing one of two things: eating or meditate. While eating, they are not thinking, and while thinking, they are not eating. The five philosophers sit at a circular table with a large bowl of rice in the center. A chopstick is placed in between each pair of adjacent philosophers and to eat he needs two chopsticks. Each philosopher can only use the chopstick on his immediate left and immediate right.

```
# let b =
    let b0 = Array.create 5 (Mutex.create()) in
     for i=1 to 4 do b0.(i) <- Mutex.create() done;
     b0 ;;
val b : Mutex.t array = [|<abstr>; <abstr>; <abstr>; <abstr>; <abstr>|]
```

Dining philosophers

```
# let meditation = Thread.delay
  and eating = Thread.delay ;;
```

```
let philosopher i =
 let ii = (i+1) mod 5
 in while true do
     meditation 3. ;
     Mutex.lock b.(i);
     Printf.printf "Philo (%d) takes his left-hand chopstick" i ;
     Printf.printf " and meditates a little while more\n";
     meditation 0.2;
     Mutex.lock b.(ii);
     Printf.printf "Philo (%d) takes his right-hand chopstick\n" i;
     eating 0.5;
     Mutex.unlock b.(i);
     Printf.printf "Philo (%d) puts down his left-hand chopstick" i;
     Printf.printf " and goes back to meditating\n";
     meditation 0.15;
     Mutex.unlock b.(ii);
     Printf.printf "Philo (%d) puts down his right-hand chopstick\n" i
    done ;;
```

Dining philosophers

We can test this little program by executing:

```
for i=0 to 4 do ignore (Thread.create philosopher i) done ;
while true do Thread.delay 5. done ;;
```

Problems:

- Deadlock: all philosophers can take their left-hand chopstick, so the program is stuck.
- Starvation: To avoid deadlock, the philosophers can put down a chopstick if they do not manage to take the second one. This is highly courteous, but still allows two philosophers to gang up against a third to stop him from eating.

Exercise

Think about solutions to avoid deadlock and starvation

Module `Condition`

- `create : unit -> Condition.t` returns a new condition variable.
- `wait : Condition.t -> Mutex.t -> unit`
  `wait c m` atomically unlocks the mutex `m` and suspends the calling process on the condition variable `c`. The process will restart after the condition variable `c` has been signaled. The mutex `m` is locked again before `wait` returns.
- `signal : Condition.t -> unit`
  `signal c` restarts one of the processes waiting on the condition variable `c`.
- `broadcast : Condition.t -> unit`
  `broadcast c` restarts all processes waiting on the condition variable `c`.

Typical usage pattern:

```
Mutex.lock m;
   while (* some predicate P over D is not satisfied *) do
     Condition.wait c m
   done;
   (* Modify D *)
   if (*  the predicate P over D is now satisfied *) then Condition.signa
Mutex.unlock m
```

Example: a Monitor

```
module SynchQueue = struct
  type 'a t =
    { queue : 'a Queue.t; lock : Mutex.t; non_empty : Condition.t }

  let create () = {
    queue = Queue.create ();
    lock = Mutex.create ();
    non_empty = Condition.create ()
  }

  let add e q =
    Mutex.lock q.lock;
    if Queue.length q.queue = 0 then Condition.broadcast q.non_empty;
    Queue.add e q.queue;
    Mutex.unlock q.lock

  let remove q =
    Mutex.lock q.lock;
    while Queue.length q.queue = 0 do
      Condition.wait q.non_empty q.lock done;
    let x = Queue.take q.queue in
    Mutex.unlock q.lock; x
end
```

Monitors

OCaml does not provide explicit constructions for monitors. They must be implemented by using mutexes and condition variables. Other languages provides monitors instead, for instance Java.

Monitors in Java:

- In Java a monitor is any object in which at least one method is declared `synchronized`
- When a thread is executing a synchronized method of some object, then the other threads are blocked if they call any synchronized method of that object.

```
class Account{
  float balance;
  synchronized void deposit(float amt) {
     balance += amt;
  }
  synchronized void withdraw(float amt) {
    if (balance < amt)
      throw new OutOfMoneyError();
    balance -= amt;
  }
}
```

Outline

What's wrong with locking

- **Locking has many pitfalls for the inexperienced programmer**

Priority inversion: a lower priority thread is preempted while holding a lock needed by higher-priority threads.

Convoying: A thread holding a lock is descheduled due to a time-slice interrupt or page fault causing other threads requiring that lock to queue up. When rescheduled it may take some time to drain the queue. The overhead of repeated context switches and underutilization of scheduling quanta degrade overall performance.

Deadlock: threads that lock the same objects in different order. Deadlock avoidance is difficult if many objects are accessed at the same time and they are not statically known.

Debugging: Lock related problems are difficult to debug (since, being time-related, they are difficult to reproduce).

Fault-tolerance If a thread (or process) is killed or fails while holding a lock, what does happen? (cf. `Thread.delete`)

- **Programming is not easy with locks and requires difficult decisions:**
  - Taking too few locks — leads to race conditions.
  - Taking too many locks — inhibits concurrency
  - Locking at too coarse a level — inhibits concurrency
  - Taking locks in the wrong order — leads to deadlock
  - Error recovery is hard (eg, how to handle failure of threads holding locks?)
- **A major problem: Composition**
  - Lock-based programs do not compose: For example, consider a hash table with thread-safe insert and delete operations. Now suppose that we want to delete one item A from table t1, and insert it into table t2; but the intermediate state (in which neither table contains the item) must not be visible to other threads. Unless the implementer of the hash table anticipates this need, there is simply no way to satisfy this requirement.

Locks are non-compositional

Consider the previous (correct) Java bank Account class:

```
class Account{
  float balance;

  synchronized void deposit(float amt) {
    balance += amt;
  }

  synchronized void withdraw(float amt) {
    if (balance < amt)
      throw new OutOfMoneyError();
    balance -= amt;
  }
}
```

Now suppose we want to add the ability to transfer funds from one account to another.

Locks are non-compositional

Simply calling withdraw and deposit to implement transfer causes a race condition:

```
class Account{
  float balance;
  ...
  void badTransfer(Acct other, float amt) {
  other.withdraw(amt);
  // here checkBalances sees bad total balance
  this.deposit(amt);
  }
}

class Bank {
  Account[] accounts;
  float global_balance;

  checkBalances () {
    return (sum(Accounts) == global_balance);
  }
}
```

Locks are non-compositional

Synchronizing transfer can cause deadlock:

```
class Account{
  float balance;

  synchronized void deposit(float amt) {
    balance += amt;
  }

  synchronized void withdraw(float amt) {
    if(balance < amt)
      throw new OutOfMoneyError();
    balance -= amt;
  }

  synchronized void badTrans(Acct other, float amt) {
    // can deadlock with parallel reverse-transfer
    this.deposit(amt);
    other.withdraw(amt);
  }
}
```

Concurrency without locks

We need to synchronize threads without resorting to locks

1. *Cooperative threading*
   The threads themselves relinquish control once they are at a stopping point.

2. *Channeled communication*
   The threads do not share memory. All data is exchanged by explicit communications that take place on channels.

3. *Software transactional memory*
   Each thread declares the blocks that must be performed atomically. If the execution of an atomic block causes any conflict, the modifications are rolled back and the block is re-executed.

Concurrency without locks

1. *Cooperative threading:* The threads themselves relinquish control once they are at a stopping point.
   Pros: Programmer manage interleaving, no concurrent access happens
   Cons: The burden is on the programmer: the system may not be responsive (eg, Classic Mac OS 5.x to 9.x). Does not scale on multi-processors. Not always compositional.
2. *Channeled communication:* The threads do not share memory. All data is exchanged by explicit communications that take place on channels.
   Pros: Compositional. Easily scales to multi-processor and distributed programming (if asynchronous)
   Cons: Awkward when threads concurrently work on complex and large data-structures.
3. *Software transactional memory:* If the execution of an atomic block cause any conflict, modification are rolled back and the block re-executed.
   Pros: Very compositional. A no brainer for the programmer.
   Cons: Very new, poorly mastered. Feasibility depends on conflict likelihood.

Concurrency without locks

Besides the previous solution there is also a more drastic solution (not so general as the previous ones but composes with them):

- *Lock-free programming*

  > *Threads access to shared data without the use of synchronization primitives such as mutexes. The operations to access the data ensure the absence of conflicts.*

  Idea: instead of giving operations for mutual exclusion of accesses, define the access operations so that they take into account concurrent accesses.

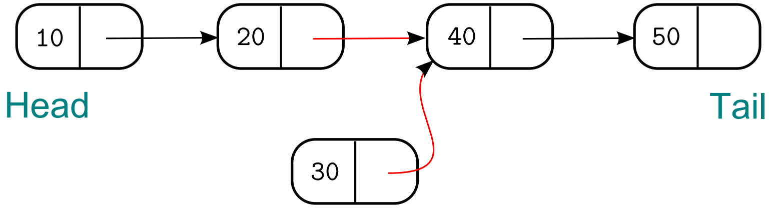
  Pros: A no-brainer if the data structures available in your lock-free library fit your problem. It has the granularity precisely needed (if you work on a queue with locks, should you use a lock for the whole queue?)

  Cons: requires to have specialized operations for each data structure. Not modular since composition may require using a different more complex data structure. Works with simple data structure but is hard to generalize to complex operations. Hard to implement in the absence of hardware support (e.g., `compare_and_swap`).

Lock-free programming: an example

Non blocking linked list:

Insertion:

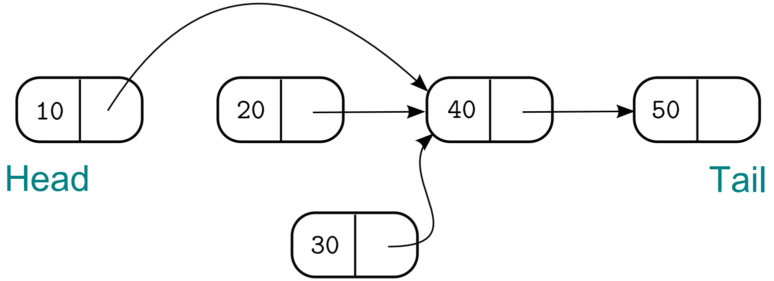


Compare and swap: compare the two links *pos* and *next* (marked in red)

- If they are the same then no conflicting operation has been performed, so atomically swap the successor of the second element with the pointer to the new element.
- Otherwise, a conflicting modification was performed: **retry the insert**.

Lock-free programming: Non blocking linked list:

Deletion is more difficult:



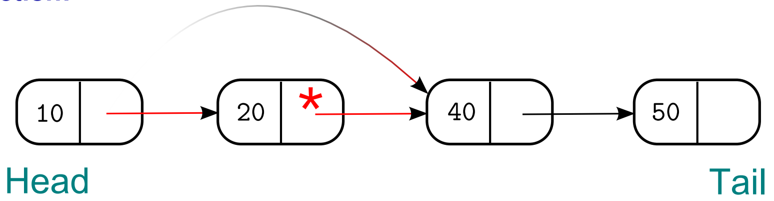
HOWEVER

If before the compare and swap another thread makes an insertion such as the above, then:

- **The compare and swap succeeds**
- **The insertion succeeds**
- **The insertion is lost**

Lock-free programming: Non blocking linked list:

Deletion:



If we want to delete the first element then we must proceed as follows:

1. **Mark the element to delete: a marked point can still be traversed but will be ignored by concurrent insertion/deletions**
2. **Record the pointer to the element next to the one to be deleted**
3. **Compare the old and new pointer to the successor of the 10 element and swap them (atomic compare and swap)**

Lock-free programming

Summary:

- Lock-free programming requires specifically programmed *data structures* while the next solutions require specific *control structures*.
- As such, it is of a less general application than the techniques we describe next.
- Also it may not fit modular development, since a structure composed of lock-free programmed data structures may fail to avoid global conflicts.
- However when it works, then it comes from free and can be combined with any of the techniques that follow, thus reducing the logical complexity of process synchronization.

Outline

Cooperative multi-threading in OCaml: `Lwt`

Cooperative multi-threading in OCaml is available by the `Lwt` module

Rationale

Instead of using few large monolithic threads, define (1) many small interdependent threads, (2) the interdependence relation between them.

- A thread executes without interrupting till a *cooperation point* where it passes the control to another thread
- A *cooperation point* is reached either by explicitly passing the control (function `yield`), or by calling a “cooperative” function (eg, `read`, `sleep`).
- `Lwt` uses a non-preemptive scheduler which is an event loop.
- At each cooperation point the thread passes the control to the scheduler, which handles it to another thread.

Nota bene

Do not call blocking functions, otherwise all threads will be blocked. In particular *do not* use `Unix.sleep` and `Unix.read`, but the corresponding cooperative versions `Lwt_unix.sleep` and `Lwt_unix.read`, instead.

`Lwt` threads

A thread computing a value of type `'a` is a value of abstract type `'a Lwt.t`

Each thread is in one of the following states:

1. *Terminated:* it successfully computed the value
2. *Suspended:* the computation is not over and will resume later
3. *Failed:* the computation failed (with an exception)

Examples:

- `Lwt.return :  'a -> 'a Lwt.t`
  It *immediately* returns a *terminated* thread whose computed value is the one passed as argument.
- `Lwt_unix.sleep :  float -> unit Lwt.t`
  It *immediately* returns a *suspended* thread that will return `()` after some time (if the scheduler reschedules it).
- `Lwt.fail :  exn -> 'a Lwt.t`
  It *immediately* returns a *failed* thread whose exception is the one passed as argument.

`Lwt` threads

`Lwt` threads are a monad:

- `Lwt.bind :  'a Lwt.t -> ('a -> 'b Lwt.t) -> 'b Lwt.t`

  The expression `Lwt.bind p f` (that can also be written as `p >>= f`) *immediately* returns a thread of type `'b Lwt.t`, defined as follows:

  - If the thread `p` is *terminated* then it passes it results to `f`.
  - If the thread `p` is *suspended* then `f` is saved in the list of the functions waiting for the result of `p`. When `p` terminates, then the scheduler actives these functions one after the other.
  - If the thread `p` is *failed*, then so is the whole expression.

Nota bene

Bind is not a cooperation point: it does not imply any suspension

Example

```
rlwrap ocaml
Objective Caml version 3.11.1
# #use "topfind";;

# #require "lwt.unix";;

# Lwt.bind (Lwt.return 3) (fun x ->
            print_int x; Lwt.return());;
3- : unit Lwt.t = <abstr>

# Lwt.bind (Lwt_unix.sleep 3.0) (fun () ->
            print_endline "hello"; Lwt.return ());;
- : unit Lwt.t = <abstr>
```

- `(Lwt.return 3) >>= (fun x -> print_int x; Lwt.return())` immediately returns a thread of type `unit Lwt.t` after having printed `3`. Notice the use of `Lwt.return ()` for well typing
- `(Lwt_unix.sleep 3.0) >>= (fun () -> print_endline "hello"; Lwt.return ())` immediately returns a thread of type `unit Lwt.t` and nothing else.

In order to see the last thread to behave as expected (and print after three seconds “hello”) we have to run the scheduler, that is the function `Lwt_unix.run : 'a Lwt.t -> 'a`

- `Lwt_unix.run t` manage a queue of waiting threads without preemption. It terminates when the thread `t` does.
- When we run the scheduler we see the computation above to end (since the schedule reactivates `Lwt_unix.sleep 3.0` which can pass its hand to the next thread (and then it ends after 30 seconds).

```
# Lwt_unix.run (Lwt_unix.sleep 30.);;
hello
- : unit = ()
```

Lwt_unix

The main function provided by Lwt_unix is run:

- It manages a queue of threads ready to be executed. As long as this queue is not empty it runs them in the order.
- It maintains a table of open file descriptors together with the threads that wait on them and insert them in the queue as soon as they have received the data they were waiting for.
- It inserts in the queue the threads that exceeded their sleep time.
- It iterates and stops when its argument thread does

Besides the scheduler Lwt_unix provides the cooperative version of most of the functions in the Unix module:

- `Lwt_unix.yield : unit -> unit Lwt.t` forces a cooperation point adding the thread in the scheduler queue.
- Lwt_unix.read. Works as Unix.read but while the latter immediately blocks, the former immediately returns a new thread which:
  - It tries to read the file
  - If data is available, then it returns a result
  - Otherwise, it sleeps in the queue associated to the file descriptor

“do” notation for `Lwt_unix`

It is possible to use the “`lwt_in`” notation to mimic Haskell’s “`do`” notation. So

```
Lwt_chan.input_line ch >>= fun s ->
Lwt_unix.sleep 3. >>= fun () ->
print_endline s;
Lwt.return ()
```

can be written as

```
lwt s = input_line ch in
lwt () = Lwt_unix.sleep 3. in
print_endline s;
Lwt.return ()
```

Example

A thread that writes “hello” every ten seconds

```
let rec f () =
print_endline "hello";
Lwt_unix.sleep 10. >>= f
in f ();
```

Join of threads

Let f and g be functions that return threads (e.g., `unit -> ‘a Lwt.t`)

```
let first_thread = f () in          // launch the first thread
let second_thread = g () in         // launch the second thread
lwt fst_result = first-thread in    // wait for the first thread result
lwt snd_result = second_thread in   // wait for the second thread result
  ⋮
```

Two versions of cooperative List.map

Two versions of map running a thread for each value in the list.

```
# let rec map f l =
    match l with
    | [] -> return []
    | v :: r ->
       let t = f v in
       let rt = map f r in
       t >>= fun v' ->
       rt >>= fun l' ->
       return (v' :: l');;
val map : ('a -> 'b Lwt.t) -> 'a list -> 'b list Lwt.t = <fun>
```

```
let rec map2 f l =
  match l with
  | [] -> return []
  | v :: r ->
      f v >>= fun v' ->
      map2 f r >>= fun l' ->
      return (v' :: l')
 val map : ('a -> 'b Lwt.t) -> 'a list -> 'b list Lwt.t = <fun>
```

Outline

Communication on channels

Two kinds of communication:

1. *Synchronous communication:*
   sending a message on an action is blocking.

   Module `Event` *of the OCaml's thread library.*

2. *Asynchronous communications*:
   sending a message is a non-blocking action: messages are buffered and the order is preserved.

   Erlang-style concurrency.

Synchronous communications: OCaml's `Event`

- `type 'a channel`
  The type of communication channels carrying values of type `'a`.
- `new_channel : unit -> 'a channel`
  Return a new channel.
- `type +'a event`
  The type of communication events returning a result of type `'a`.
- `send : 'a channel -> 'a -> unit event`
  `send ch v` returns the event consisting in sending the value `v` over the channel `ch`. The result value of this event is `()`.
- `receive : 'a channel -> 'a event`
  `receive ch` returns the event consisting in receiving a value from the channel `ch`. The result value of this event is the value received.
- `choose : 'a event list -> 'a event`
  `choose evl` returns the event that is the parallel composition of all the events in the list `evl`.

Synchronous communications: OCaml's `Event`

The functions `send` and `receive` are not blocking functions

The primitives `send` and `receive` build the elementary events "sending a message" or "receiving a message" but do not have an immediate effect.

- They just create a data structure describing the action to be done
- To make an event happen, a thread must synchronize with another thread wishing to make the complementary event happen
- The `sync` primitive allows a thread to wait for the occurrence of the event passed as argument.
  - `sync :  'a event -> 'a`
    "Synchronize" on an event: offer all the communication possibilities specified in the event to the outside world, and block until one of the communications succeeds. The result value of that communication is returned.
  - `poll :  'a event -> 'a option`
    Non-blocking version of `Event.sync`: offer all the communication possibilities specified in the event to the outside world, and if one can take place immediately, perform it and return `Some r` where `r` is the result value of that communication. Otherwise, return `None` without blocking.

Example: access a shared variable via communications

```
# let ch = Event.new_channel () ;;
# let v = ref 0;;

# let reader () = Event.sync (Event.receive ch);;
# let writer () = Event.sync (Event.send ch ("S" ^ (string_of_int !v)));;

# let loop_reader s d () = (
    for i=1 to 10 do
     let r = reader() in print_string (s ^ ":" ^ r ^ "; "); flush stdout;
     Thread.delay d
    done;
    print_newline());;
# let loop_writer d () =
    for i=1 to 10 do incr v; writer(); Thread.delay d done;;

# Thread.create (loop_reader "A" 1.1) ();;
# Thread.create (loop_reader "B" 1.5) ();;
# Thread.create (loop_reader "C" 1.9) ();;

# loop_writer 1. ();;
A:S1; C:S2; B:S3; A:S4; C:S5; B:S6; A:S7; C:S8; B:S9; A:S10;
- : unit = ()
# loop_writer 1. ();;
C:S11; B:S12; A:S13; C:S14; B:S15; A:S16; C:S17; B:S18; A:S19; C:S20;
- : unit = ()
```

Example: Single position queue

We modify the Queue example so as mutual exclusion is obtained by channels rather than mutexes. We keep the same interface

```
module Cell = struct
  type 'a t =
    { add_ch: 'a Event.channel; rmv_ch: 'a Event.channel }

  let create () =
    let aCh = Event.new_channel ()
    and rCh = Event.new_channel () in
    let rec empty () =
      let e = Event.sync (Event.receive aCh) in
      full e
    and full e =
      empty (Event.sync (Event.send rCh e))
    in
      ignore (Thread.create empty ());
      {add_ch = aCh ; rmv_ch = rCh}

   let add e q =
     Event.syncpoll(Event.send q.add_ch e)

   let remove q =
     Event.sync (Event.receive q.rmv_ch)
end
```

Asynchronous communications: Erlang's concurrency

Erlang is a (open-source) general-purpose concurrent programming language and runtime system designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications.

- The sequential subset of Erlang is a strict and dynamically typed functional language. For concurrency it follows the Actor model. Three primitives
    - `spawn` spawns a new process
    - `send` asynchronously send a message to a given process
    - `receive` reads one the of the received messages
- Concurrency is structured around *processes*.
  Erlang processes are are not OS processes: they are much lighter (scale up to hundreds of million of processes). Like OS processes and unlike OS threads or green threads they have no shared state between them.
- Process communication is done via *asynchronous message passing*: every process has a "mailbox", a queue of messages that have been sent by other processes and not yet consumed.

- Process creation
  ```
  Pid = spawn(Module, FunctionName, ArgumentList)
  ```
  Spawns a new process that executes the function `FunctionName` in the module `Module` with arguments `ArgumentList` and returns *immediately* its identifier.
- Asynchronous send
  ```
          Pid ! Message
  ```
  Put the message `Message` in the buffer of the process whose identifier is `Pid`. So `foo(12) !  bar(baz)` will first evaluate `foo(12)` to get the process identifier and `bar(baz)` for the message to send, and returns *immediately* (the value of the message) without waiting for the message either to arrive at the destination or to be received.
- Selective receive
  ```
  receive
     Pattern1  -> Actions1 ;
     Pattern2  -> Actions2 ;
         ⋮
  end
  ```
  Select in the mailbox the first message that matches a pattern, remove it from the mailbox, and execute its actions. Otherwise suspend.

An example

```
% Create a process and invoke the function
% web:start_server(Port, MaxConnect)
ServerProcess = spawn(web, start_server, [Port, MaxConnect]),

% [Distribution support]
% Create a remote process and invoke the function
% web:start_server(Port, MaxConnect) on machine RemoteNode
RemoteProcess = spawn(RemoteNode, web, start_server, [Port, MaxConnect]),

% Send a message to ServerProcess (asynchronously). The message consists
% of a tuple with the atom "pause" and the number "10".
ServerProcess ! pause, 10,

% Receive messages sent to this process
receive
       data, DataContent -> handle(DataContent);
       hello, Text -> io:format("Got hello message: ~s", [Text]);
       goodbye, Text -> io:format("Got goodbye message: ~s", [Text])
end.
```

Erlang-style concurrency

This style of concurrency has been adopted in several other languages

- F#: *cf.* the `MailboxProcessor` class
- Scala: uses the same syntax (and semantics) as Erlang but instead of processes we have “actor objects” that run in separate threads.
- Retlang a Erlang inspired library for .NET and `Jetlang` its Java counterpart.
- Others: Termite Scheme, Coro Module for Perl, ...

Outline

Compositionality

The main problem with locks is that they are not compositional

```
action1 = withdraw a 100
```

```
action2 = deposit b 100
```

```
action3 =
   do withdraw a 100
       Inconsistent state
      deposit b 100
```

Solution: Expose all locks

```
action3 =
   do lock a
      lock b
      withdraw a 100
      deposit b 100
      release a
      release b
```

Problems

- Risk of deadlocks
- Unfeasible for more complex cases

Software transactional memory

Idea

- Borrow ideas from database people
  - ACID transactions:
    - **atomic** - all changes commit or all changes roll back; changes appear to happen at a single moment in time
    - **consistent** - operate on a snapshot of memory using newest values at beginning of the txn
    - **isolated** - changes are visible to other threads, only after commit and, viceversa, threads in a transaction cannot see other thread changes.
    - **durable** - does not apply to STM (changes do not persist on disk) but we can adapt it when changes are lost if software crashes or hardware fails (cf. locks).
- Add ideas from (pure) functional programming
  - Computations are first-class values
  - What side-effects can happen and where they can happen is controlled

Software Transactional Memory

- First ideas in 1993
- New developments in 2005
  - (Simon Peyton Jones, Simon Marlow, Tim Harris, Maurice Herlihy)

Atomic transactions

Write sequential code, and wrap `atomically` around it

```
action3 =
   atomically{
      withdraw a 100
      deposit b 100
   }
```

How does it works?

- Execute body without locks
- Each memory access is logged to a thread-local transaction log.
- No actual update is performed in memory
- At the end, we try to *commit* the log to memory
- Commit may fail, then we retry the whole atomic block

Optimistic concurrency

Caveats

Simon Peyton-Jones's missile program:

```
action3 =
atomically{
   withdraw a 100
   launchNuclearMissiles
   deposit b 100
}
```

No side effects allowed!

More in details:

The logging capability is implemented by specific "transactional variables".

Absolutely forbidden:

- To read a transaction variable outside an atomic block
- To write to a transaction variable outside an atomic block
- To make actual changes (eg, file or network access, use of non-trasactional variables) inside an atomic block...

These constraints are enforced by the type system

STM in Haskell

- Fully-fledged implementation of STM: `Control.Concurrent.STM`
- Implemented in the language also in Clojure and Perl6.
- Implementations for C++, Java, C#, F# being developed as libraries ... difficult to solve all problems
- In Haskell, it is easy: controlled side-effects

```
type STM a
instance Monad STM
    atomically  :: STM a -> IO a
    retry       :: STM a
    orElse      :: STM a -> STM a -> STM a

type TVar a
    newTVar     :: a -> STM (TVar a)
    readTVar    :: TVar a -> STM a
    writeTVar   :: TVar a -> a -> STM ()
```

Sides effects must be performed on specific “transactional variables” `TVar`

STM: TVar

Threads in STM Haskell communicate by reading and writing transactional variables

```
type Resource = TVar Int

putR :: Resource -> Int -> STM ()
putR r i = do v <- readTVar r
              writeTVar r (v+i)

main = do ... atomically (putR r 13) ...
```

Operationally, `atomically` takes the tentative updates and actually applies them to the `TVars` involved. The system maintains a per-thread transaction log that records the tentative accesses made to `TVars`.

STM: `retry`

`retry`: Retries execution of the current memory transaction because it has seen values in TVars which mean that it should not continue (e.g., the TVars represent a shared buffer that is now empty). The implementation may block the thread until one of the TVars that it has read from has been updated.

```
retry :: STM a

atomically {if n_items == 0 then retry
                else ...remove from queue...}
```

In summary:

- `retry` says "abandon the current transaction and re-execute it from scratch"
- The implementation *waits* until `n_items` changes
- No *condition variables*, no lost wake-ups!

Blocking composes

```
atomically { x = queue1.getItem()
           ; queue2.putItem( x ) }
```

- If either `getItem` or `putItem` retries, the whole transaction retries
- So the transaction waits until queue1 is not empty AND queue2 is not full
- No need to re-code getItem or putItem
- (Lock-based code does not compose)

STM `orElse`

`orElse`: it tries two alternative paths:

- If the first retries, it runs the second
- If both retry, the whole `orElse` retries.

```
orElse :: STM a -> STM a -> STM a

atomically { x = queue1.getItem();
                queue2.putItem(x)
           `orElse`
                queue3.putItem(x) }
```

- So the transaction waits until
  - queue1 is non-empty, AND
  - EITHER queue2 is not full OR queue3 is not full

  without touching `getItem` or `putItem`

Note:

```
m1 `orElse` (m2 `orElse` m3) = (m1 `orElse` m2) `orElse` m3
            retry `orElse` m = m
            m `orElse` retry = m
```

Compositionality

- All transactions are flat
- Calling transactional code from the current transaction is normal
- This simply extends the current transaction

STM in Haskell

- Safe transactions through type safety
  - A very specific monad STM (distinct from I/O)
  - We can only access TVars
  - TVars can only be accessed in STM monad
  - Referential transparency inside blocks
- Explicit retry – expressiveness
- Compositional choice – expressiveness

Problems

- Overhead: managing transactions bookkeeping requires some overhead
- Starvation: could the system “thrash” by continually colliding and re-executing?
  - No: one transaction can be forced to re-execute only if another succeeds in committing. That gives a strong progress guarantee.
  - But a particular thread could perhaps starve.
- Performance: potential for many retries resulting in wasted work
- Tools: support is currently lacking
  - for learning which memory locations experienced write conflicts
  - for learning how often each transaction is retried and why

Problems in C++/Java/C#

- Retry semantics
- IO in atomic blocks
- Access of transaction variables outside of atomic blocks
- Access to regular variables inside of atomic blocks

A useful analogy

Switch from manual to automatic gear:

- **Memory management**
  - malloc, free, manual refcounting
  - Garbage collection
- **Concurrency**
  - Mutexes, semaphores, condition variables
  - Software transactional memory

Manual/auto tradeoffs

Same kind of tradeoffs:

- **Memory management**
  - Manual: Performance, footprint
  - Auto: Safety against memory leaks, corruption
- **Concurrency**
  - Manual: Fine tuning for high contention
  - Auto: Safety against deadlock, corruption

Rationale

In both cases you pay in terms of performance and of “I do not quite know what is going on”, but they allow you to build larger, more complex systems that won’t break because of wrong “by hand” management.