TÖL304G
Forritunarmál — Programming Languages
Vikublað 6 — Weekly 6

Snorri Agnarsson

21. september 2024

CAML Light, Objective CAML

Við munum kíkja á forritunarmálin CAML Light og Objective CAML, sem eru afbrigði af CAML, sem aftur er afbrigði af ML forritunarmálinu. Í framhaldinu munum við yfirleitt ekki gera greinarmun á CAML, CAML Light og Objective CAML.

We will take a look at the programming languages CAML Light and Objective CAML, which are variants of CAML, which in turn is a variant of the ML programming language.

Það sem einkennir ML og CAML er tögunin (*typing*) í þeim. Þau eru rammtöguð (*strongly typed*) og nota sömu tögunaraðferð, sem er sérstök að því leyti að þýðandinn sér mikið til um að finna út úr því hvert tagið á að vera á hinum magvíslegustu gildum og segðum.

What distinguishes ML and CAML is the typing in them. They are strongly typed and use the same typing method, which is distinguished by the fact that the compiler mostly handles figuring out what the type should be for the various values and expressions.

Til dæmis má nota eftirfarandi forritstexta í CAML til að skilgreina fall fyrir $n!$:

For example we can use the following CAML code to define a function for $n!$:

```
let rec fact n =
    if n=0 then
        1.0
    else
```

```
        (float_of_int n) *. (fact (n-1))
;;
```

CAML þýðandinn mun lesa þessa skilgreiningu og draga þá ályktun að fallið `fact` sé af tagi `int -> float`. Þýðandinn sér að viðfangið `n` er af tagi `int` vegna þess að fallinu `float_of_int` er beitt á það, sem vitað er að tekur `int` sem viðfang (og skilar `float`), og útkoman úr `fact` er af tagi `float` vegna þess að lesfastinn `1.0` er af tagi `float` og fallið `*.` skilar ávallt `float`.

The CAML compiler will read this definition and deduce that the function `fact` is of type `int -> float`. The compiler sees that the argument `n` is of type `int` because the function `float_of_int` is applied to it, and it knows that `float_of_int` takes a value of type `int` as an argument (and returns a value of type `float`). And the return value from `fact` is of type `float` because the literal `1.0` is of type float and the function `*.` always returns a value of type `float`.

Ef við skrifum fallið svona

If we write the function like this

```
let rec fact n =
    if n=0 then
        1
    else
        (float_of_int n) *. (fact (n-1))
;;
```

kvartar þýðandinn yfir því að fallið skili `int` (lesfastinn `1`) á einum stað, en `float` á öðrum stað. Þýðandinn leyfir það ekki.

the compiler will complain that the function returns an `int` (the literal `1`) in one place, but returns a `float` in another place. The compiler does not allow that.

Öll föll í CAML taka *nákvæmlega eitt* viðfang af einhverju vel skilgreindu tagi og skila einu gildi af vel skilgreindu tagi. Þó er mikilvægt að hafa eitt lykilatriði í huga: Tagskilgreiningar í CAML (og ML) geta innihaldið *frjálsar tagbreytur*. Til dæmis er fallið `hd`, sem samsvarar `car` í LISP, af tagi `list 'a -> 'a`. Þetta þýðir að fallið tekur viðfang af tagi `list 'a`, þar sem `'a` stendur fyrir hvaða tag sem er, og skilar þá gildi af tagi `'a`.

All functions in CAML take *exactly one* argument of some well defined type and return one value of a well defined type. However, it is important to understand one key issue: the type definitions in CAML (and ML) can contain *free type variables*. For example the function `hd`, which corresponds to `car` in LIST, is of type `list 'a -> 'a`. This means that the function takes an argument of type `list 'a`, where `'a` stands for any type, and then returns a value of type `'a`.

Vegna tögunarinnar er í CAML gerður greinarmunur á pörum og listum, sem ekki er gert í Morpho og LISP (þ.m.t. Scheme), ef viðkomandi gildi er ekki tómi listinn. Ef til dæmis `'a` og `'b` eru tvö tög þá er `'a * 'b` tag para þar sem fyrra stakið er af tagi

`'a` og seinna af tagi `'b`. Aftur á móti er `list 'a` tag lista gilda af tagi `'a`. Slíkur listi má vera tómur, en það getur par ekki verið.

Because of the typing a distinction is made in CAML between pairs and lists, which is not done in Morpho and LISP (including Scheme), if the value in question is not the empty list. If, for example, `'a` and `'b` are two types then `'a * 'b` is the type of pairs where the first value is of type `'a` and the second of type `'b`. On the other hand `list 'a` is the type of lists of values containing items of type |'a|. Such a list can be empty, but a pair can not be empty.

CAML má finna á vefsíðum frönsku rannsóknarstofnunarinnar INRIA[1]. Þar má fá CAML í ýmsum útgáfum og útfærslum, bæði CAML Light og Objective CAML, sem nú heitir OCAML, fyrir ýmis stýrikerfi. Mælt er með CAML Light fyrir þetta námskeið, en ef menn ætla að nota CAML fyrir bitastæð verkefni er næstum örugglega betra að nota OCAML. Þeir sem áhuga hafa mega vel nota OCAML í námskeiðinu, í stað CAML Light.

CAML can be found on the web pages of the French research institute INRIA[2]. There you can find CAML in various versions and implementations, both CAML Light and Objective CAML, now called OCAML, for various operating systems. CAML Light is recommended for this course, but if people want to use CAML for industrial purposes it is almost certainly better to use OCAML. Those who are interested can use OCAML in this course instead of CAML Light.

Hvar er CAML Light? Where is CAML Light?

CAML Light kerfið fyrir ýmis stýrikerfi má sækja af vefnum[3].

The CAML Light system for various operating system can be fetched from the web[4].

Nokkur CAML Light föll — A Few CAML Light Functions

Athugið að flest þessi föll eru reyndar einnig innbyggð í CAML Light.

Note that most of these functions are in fact built-in in CAML Light.

Haus lista — The Head of a List

```
(*
```

[1] http://caml.inria.fr/download.en.html

[2] http://caml.inria.fr/download.en.html

[3] http://caml.inria.fr/caml-light/index.en.html

[4] http://caml.inria.fr/caml-light/index.en.html

```
** Notkun: hd x
** Fyrir:  x er listi, ekki tómur
** Gildi:  Hausinn á x, þ.e. fremsta gildið

** Usage:  hd x
** Pre:    x is a list, not empty
** Value:  The head of x, i.e. the frontmost value
*)
let hd x =
  match x with
    [] ->
      raise
        (Invalid_argument
          "attempt to take head of empty list"
        )
  |
    a::b ->
      a
;;
(* hd : 'a list -> 'a = <fun> *)
```

Hali lista — The Tail of a List

```
(*
** Notkun: tl x
** Fyrir:  x er listi, ekki tómur
** Gildi:  Halinn á x

** Usage:  tl x
** Pre:    x is a list, not empty
** Value:  The tail of x
*)
let tl x =
  match x with
    [] ->
      raise
        (Invalid_argument
          "attempt to take tail of empty list"
        )
  |
    a::b ->
      b
```

```
;;
(* tl : 'a list -> 'a list = <fun> *)
```

Innsetning frá vinstri — Folding from the Left

```
(*
Notkun: it_list f u x
Fyrir:  f er tvíundaraðgerð, f: A->B->A,
        u er gildi af tagi A,
        x=[x1;...;xN] er listi gilda af
        tagi B.
Gildi:  u+x1+x2+...+xN, reiknað frá vinstri
        til hægri, þar sem a+b = (f a b).

Usage:  it_list f u x
Pre:    f is a binary operation, f: A->B->A,
        u is a value of type A,
        x=[x1;...;xN] is a list of values of
        type B.
Value:  u+x1+x2+...+xN, computed from left
        to right, where a+b = (f a b).
*)
let rec it_list f u x =
  if x=[] then
    u
  else
    it_list f (f u (hd x)) (tl x)
;;

(*
it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
*)
```

Innsetning frá hægri — Folding from the Right

```
(*
Notkun: list_it f x u
Fyrir:  f er tvíundaraðgerð, f: A->B->B,
        u er gildi af tagi B,
        x=[x1;...;xN] er listi gilda af
        tagi A.
Gildi:  x1+x2+...+xN+u, reiknað frá hægri
        til vinstri, þar sem a+b = (f a b).
```

```
Usage:  list_it f x u
Pre:    f is a binary operation, f: A->B->B,
        u is a value of type B,
        x=[x1;...;xN] is a list of values of
        type A.
Value:  x1+x2+...+xN+u, computed from right
        to left, where a+b = (f a b).
*)
let rec list_it f x u =
  if x=[] then
    u
  else
    f (hd x) (list_it f (tl x) u)
;;

(*
list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
*)
```

Beiting falls á lista — Mapping a Function on a List

```
(*
** Notkun: map f x
** Fyrir:  x er 'a list = [x1;x2;...;xN], f er 'a->'b
** Gildi:  listinn [f x1;f x2;...;f xN]

** Usage:  map f x
** Pre:    x er 'a list = [x1;x2;...;xN], f is 'a->'b
** Value:  The list [f x1;f x2;...;f xN]
*)
let rec map f x =
  if x = [] then
    []
  else
    (f (hd x)) :: (map f (tl x))
;;
(* map : ('a -> 'b) -> 'a list -> 'b list = <fun> *)
```

Viðsnúningur lista — Reversing a List

```
(*
** Notkun: reverse x
```

```
** Fyrir:  x er listi [x1;x2;...;xN]
** Gildi:  Listi sömu gilda og x inniheldur,
**         en í viðsnúinni röð, þ.e.
**         [xN;...;x2;x1]

** Usage:  reverse x
** Pre:    x is a list [x1;x2;...;xN]
** Value:  A list of the same values that x
**         contains, but in the reverse order
**         i.e. [xN;...;x2;x1]
*)
let reverse x =
  (*
  ** Notkun: revapp x y
  ** Fyrir:  x er listi [x1;x2;...;xN]
  **         y er listi [y1;y2;...;yM]
  **         gilda af sama tagi og x
  ** Gildi:  Listi sömu gilda og x inniheldur,
  **         en í viðsnúinni röð, með y
  **         viðskeytt, þ.e.
  **         [xN;...;x2;x1;y1;y2;...;yM]

  ** Usage:  revapp x y
  ** Pre:    x is a list [x1;x2;...;xN]
  **         y is a list [y1;y2;...;yM]
  **         of values of the same type as x
  ** Value:  A list of the values in x, in the
  **         reverse order, with y appended, i.e.
  **         [xN;...;x2;x1;y1;y2;...;yM]
  *)
  let rec revapp x y =
    if x = [] then
      y
    else
      revapp (tl x) ((hd x) :: y)
  in
    revapp x []
;;
(* reverse : 'a list -> 'a list = <fun> *)
```

Samskeyting lista — Appending Lists

```
(*
** Notkun: append x y
** Fyrir:  x er listi [x1;x2;...;xN]
**         y er listi [y1;y2;...;yM]
**         gilda af sama tagi og x
** Gildi:  Listi sömu gilda og x inniheldur,
**         með y viðskeytt, þ.e.
**         [x1;x2;...;xN;y1;y2;...;yM]

** Usage:  append x y
** Pre:    x is a list [x1;x2;...;xN]
**         y is a list [y1;y2;...;yM]
**         of values of the same type as x
** Value:  A list of the same values as x
**         contains, with y appended, i.e.
**         [x1;x2;...;xN;y1;y2;...;yM]
*)
let rec append x y =
  if x=[] then
    y
  else
    (hd x) :: (append (tl x) y)
;;
(* append : 'a list -> 'a list -> 'a list = <fun> *)
```

Y fallið — The Y Combinator

```
(*
** Notkun: y f
** Fyrir:  f er fall sem tekur lélegt fall af
**         tagi 'a -> 'b sem viðfang og skilar
**         betra falli af sama tagi
** Gildi:  Besta fall 'a -> 'b sem unnt er að
**         fá með því að endurbæta gagnslaust
**         fall með f

** Usage:  y f
** Pre:    f is a function that takes bad function
**         of type 'a -> 'b as an argument and returns
**         a better function of the same type
** Value:  The best function 'a -> 'b that can be
**         gotten by improving a useless function
```

```
**          with f
*)
let rec y f x =
  (f (y f)) x
;;
(* y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun> *)
```

Dæmi um notkun Y — An Example of Using Y

Eftirfarandi segð reiknar 10! með því að nota endurbætingarfall.
The following expression computes 10! by using an improvement function.

```
(*
** Notkun: bf f
** Fyrir:  f er fall af tagi int->float
**         þ.a. til er einhver heiltala m>=0
**         þ.a. fyrir öll n með 0<=n<m skilar
**         (f n) gildinu n!.
** Gildi:  fall g þ.a. fyrir öll n með 0<=n<=m
**         skilar (g n) gildinu n!.

** Usage:  bf f
** Pre:    f is a function of type int->float
**         such that there exists some integer
**         m>=0 such that, for all n such that
**         0<=n<m, (f n) returns the value n!.
** Gildi:  A function g such that for all n
**         with 0<=n<=m the call (g n) returns
**         the value n!.
*)
let bf f n =
  if n = 0 then
    1.0
  else
    (float_of_int n) *. f(n-1)
in
  (Y bf) 10
;;
(* - : float = 3628800 *)

(* Einnig mætti skrifa eftirfarandi. *)
(* We can also write the following.  *)
(y
```

```
 (fun f ->
    fun n ->
       if n=0 then
          1
       else
          n*(f(n-1))
 )
) 10;;
```