Formal languages and Compilers

Pointers and Dynamic heap

Walter Da Col - mat.120201

03.06.2011

Contents

1 Objective

This project aims to extend **créme CAraMeL** with pointers and dynamic memory (heap).
These two features have to be implemented as follows:

- **Pointers:**
  - declarations of variables with type *pointer to basic type*.
    (eg. var p: ^int)
  - referencing an identifier with "@" will return its location in the environment
  - access to pointed location must be done with the *dereference* operator " ^ "
- **Dynamic memory:**
  - a memory separated from the one currently implemented (that is represented with the stack where local variables of programs and subprograms are allocated) in which it is possible to allocate objects at run-time
  - deallocation of memory can be done with two methods: *Reference Counter* or *Garbage Collector*.

2 Features definition

In this section I'll explain what I've chooise to do for this project and what I've to keep in mind when I'll write down code.
Here I'll define how things works with pointers and heap:

- **Pointer:**
  - pointers can be only of basic type (int | float), pointers to data structures like Vectors are not required.
  - pointer identifier is defined in the environment and its first location will be stored in *store* (stack). In this way we can always use "@" operator to obtain pointer location (like normar Var) and "p" for its value (that will be a location).
  - declaration of pointer will set *NULL* as first location reachable by pointer.
  - no partial allocation or changes to pointer depth are allowed (if a pointer is declared as ^^int it will always contains a double reference to a location that reach an int type)
  - no "casting" or changes in pointer type (not required).
  - only item with the same "depth" can be used for assignment (see explanation).
  - pointer are used directly, so no vector of pointers or similar are allowed (not required).
- **Dynamic memory (heap):**
  - I choose Reference Counter for managing dynamic memory.
  - every "cells" are defined by a locations, a value and its reference counter.
  - heap will be defined by a list of free location and by a function $location \rightarrow heap\_entry$.
  - when a cell reach zero reference its location will be reinserted in free location list and access to this cell will be denied.
  - I keep reference modification splitted from heap operation to have more control over it.
  - only commands in CC can modify heap so *exec* function will returns a modified heap.
  - heap will have a defined size (set to 256 for no specific reason).
  - like we see on lectures, assignment on pointer will change reference count.

In pointer section I've speaked of **depth**, when I was searching for a good method to check if an assignment can be done I ended up thinking about distance from pointer destination.
If I declare a pointer with ^^int its final value is two locations distant from pointer starting location, so when I speak of "depth" inside this report I mean how many dereference I must apply to reach pointer destination.
(eg. with var p: ^^int , a pexp of the shape ^p has depth (2-1) = 1).
Depth can be found for other aexp (eg. N(1) has depth 0) and I use it for comparing pexp or for checking assigment.
(eg. with var p: ^^int , this ^p := 5; cannot be done because ^p has depth (2-1) = 1 and 5 has depth 0).
It's not a very clear approach and leads often to silly situation (mentally) but this has been clear only when I reach the end of project, at the beginning I was happy of this solution.

3 Modifications

In this section I'll analyze every file of the interpreter explaining most of the modification done to it.
We'll see tokens, syntax tree and semantics rules applied to the new features.
For syntax and general ideas of how to do things I freely take inspiration from Obj-C (reference counter) and C (pointer) but for keeping things smaller and simpler I've done some assumptions and simplifications (like Alloc(p) instead of p = (int *)malloc(sizeof(int))).

3.1 lexer.mll

Inside lexer I've created three reserved words for allocation, deallocation and for NULL location (eg. `p := nil`) plus two symbol to references and dereferences an identifier.

```
    ...
| "alloc" { ALLOC }
| "free" { FREE }
| "nil" { NIL }
    ...
| "@" { REF }
| "^" { DEREF }
```

3.2 parser.mly

I've defined an expression type (*pexp*) that matches any occurence of ^ alog with an identifier.
This expression is used in *aexp* and in *lexp* because this pattern appears both on the left and the right side of an assignment.
A further needed modification similar to the previous ones, is the definition of *ptype*, like *pexp* it has any occurence of ^ but a *btype* (int | float) as follower.
Last change I've made was adding *Alloc* and *Free* for allocation and deallocation of memory. Compared to C, in which you have to assign an allocation of type to a pointer, my implementation "allocates" and assign a location at the same time (Alloc(p)). This because in CC we don't have "casting" or generic "void" pointer so we can allocate memory based on "depth" and type of pointers.

```
    ...
%token PROGRAM PROCEDURE VAR CONST INT FLOAT BEGIN END
%token IF THEN ELSE WHILE DO FOR TO REPEAT UNTIL WRITE
%token ALLOC FREE
%token ARRAY OF LBRACKET RBRACKET DOTS CALL

    ...

%token REF DEREF NIL

%token LP RP

%token EOF
    ...
pType
 :  DEREF bType { Ptr($2) }
 | DEREF pType { PPtr($2) }
```

```
 ;

gType
 :  bType { Basic($1) }
 | ARRAY LBRACKET NAT DOTS NAT RBRACKET OF bType { Vector($8,$3,$5) }
 | pType { Pointer($1) }
 ;
    ...
cmd
    ...
 | ALLOC LP ide RP { Alloc($3) }
 | FREE LP ide RP { Free($3) }
 | CALL ide LP opt_aexp_list RP { PCall($2,$4) }
 ;
    ...
pexp
 :  DEREF ide { BaseDeref($2) }
 | DEREF pexp { PtrDeref($2) }

lexp
 :  ide { LVar($1) }
 | ide LBRACKET aexp RBRACKET { LVec($1,$3) }
 | pexp { LPointer($1) }
 ;
    ...
aexp_factor
    ...
 | pexp { IdDeref($1) }
 | REF ide { IdRef($2) }
 | NIL { Nil }
 ;
```

3.3 syntaxtree.ml

```
(* pointer expression *)
type pexp =
     BaseDeref of ide (* ^p *)
   | PtrDeref of pexp (* ^..^p *)

(* arithmetical expressions *)
type aexp =
    ...
   | IdDeref of pexp (* ^..^ide *)
   | IdRef of ide (* @ide *)
   | Nil (* NULL *)

(* left expressions *)
type lexp =
    ...
   | LPointer of pexp

(* declarations *)
type pType =
     Ptr of bType (* ^int *)
   | PPtr of pType (* ^(^..^int) *)

type gType =
    ...
   | Pointer of pType

(* commands *)
type cmd =
    ...
   | Alloc of ide
   | Free of ide
```

3.4 interpreter.ml

Most modifications of CC are done inside this file:

Exception:

```
(* pointer *)
exception PTR_NULL_POINTER of string
exception PTR_ASSIGN of string
exception PTR_DEREF of string
(* heap *)
exception NO_HEAP
exception HEAP_OUT_OF_MEM
exception STATIC_DEALLOC
(* warning suppressor - for impossible match case *)
exception WARNING_SUPPRESSOR
```

New exceptions for managing error caused by pointers and heap.

Memory:

```
type loc =
     Loc of int
   | HLoc of int (* Heap pointer *)
   | NULL (* Null location *)
```

Location has been extended to contain *heap location* and a void location (*NULL*).

```
type value =
    ...
    | ValueLoc of loc (* Pointers have a loc for value *)
```

Value can be a location in case of pointers.

```
type env_entry =
    ...
    | Descr_Pointer of loc * int * bType (* loc, depth and type *)
```

Pointers are defined in the environment like other variables but they keep trace of location (in store), depth (occurency of ^) and type.

```
type heap_entry = int * value (* reference counter * value *)
type free_loc = loc list (* list of free location *)
type heap_store = loc -> heap_entry
type heap = free_loc * heap_store
```

Definition of dynamic memory, *heap* is a couple formed by a list of free location and a function that associate a location to an entry.
Every entry contains a value and a *reference counter* for managing allocation/deallocation of memory.

```
let init_heap_store ( x : loc ): heap_entry = raise NO_HEAP
```

Initialize heap_store function ("memory cells").

```
let rec init_free_loc ( size : int ): free_loc
```

Initialize list of free heap location, size is used to define heap number of cells (default 256).

```
let new_heaploc ( fl : free_loc ):( free_loc * loc )
```

Return first location avaible and remove it from free_loc, if no location exists raise an error (OUT_OF_HEAP).

`let new_heapentry (( hs : heap_store ),( addr : loc )): heap_store`

Add to heap_store $addr \rightarrow ValueLoc(NULL)$.

`let initheap ( size : int ): heap = ( init_free_loc ( size ), init_heap_store )`

Used to set up a new heap (use previously defined functions).

`let new_heapmem ( h : heap ):( heap * loc )`

Take a free location and allocate a new cell, returns a modified heap and the location previously taken.

`let remove_mem (( h : heap ),( x : loc )): heap`

Add a location to free_loc and returns modified heap.

`let update_heapentry (( hs : heap_store ),( addr : loc ),( v : value )): heap_store`

Modify an heap_entry, used to update a value given its location.

`let update_heapentry_ref (( hs : heap_store ),( addr : loc ),( r : int )): heap_store`

Similar to previous one but modify reference counter.

`let href (( h : heap ),( x : loc )): int`

Read reference counter of a location.

`let href_up (( h : heap ),( x : loc )): heap`

Increase reference counter.

`let rec href_down (( h : heap ),( x : loc )): heap`

Decrease reference counter. This function is recursive because when a cell drop to zero reference its removed but when a cell is removed other cells pointed by it loses one pointer so we call recursivly this function to them.

`let hread (( h : heap ),( x : loc )): value`

Read memory cell at given location (s(loc) for store).

`let hwrite (( h : heap ),( x : loc ),( v : value )): heap`

Write value in memory cell at given location (returns modified heap).

Expression evaluation:

Lot of functions have been adjusted in order to fulfil with the assignment:

- *ValueLoc(_)* -> every match on value type accept ValueLoc if needed.
- *Heap* -> some functions can modify heap so these functions will return another heap, other functions instead can only read values from heap so heap can be passed as argument without returning it

If a function is modified only to meet above requirement it will not be explained or reported in this section (or better, only if it's relevant in some way).

`let rec get_pexp_dep ( e : pexp )( r : env ): int`

Evaluate depth of a given pexp (see depth explanation for details about depth).

`let rec get_pexp_id ( e : pexp ): ide`

Return identifier associated with a pexp (eg. $\hat{}\hat{}\hat{}p \rightarrow p$).

`let get_pvalue_at_depth ( i : ide )( r : env )( s : store )( h : heap )( d : int ): value`

Retrieve value pointed by a pointer at "d" depth, depth is not the number of dereference applied to identifier but a distance from it last value. So if we want know last pointed location of p we must pass 0 as "d" argument.

`let rec eval_pexp ( e : pexp ) ( r : env ) ( s : store ) ( h : heap ): value`

Evaluate a pexp and perform some check about NULL dereference.

```
let rec eval_aexp (e:aexp)(r:env)(s:store)(h:heap): value
    ...
  | Ident(i)->(
            match r(i) with
                Var(l)-> s(l)
              | Val(v)-> v
              | Descr_Pointer(p,_,_)-> s(p)
              | _ -> raise SYNTAX
  | IdRef(i)->( (* @ide *)
            match r(i) with
                Var(l)-> ValueLoc(l)
              | Descr_Pointer(p,_,_)-> ValueLoc(p)
              | _ -> raise SYNTAX
           )
  | IdDeref(exp)->(eval_pexp exp r s h) (* ^..^ide *)
  | Nil -> ValueLoc(NULL)
```

Modification to *eval_aexp* to match pointers, @identifier, pexp and Nil.
Heap is used to evaluate expression.

```
let rec get_aexp_dep (e:aexp)(r:env)(s:store)(h:heap): int
```

Evaluate "depth" of a generic aexp, is used with `get_pexp_dep` to check if an assignment can be done.

Type checking and utility:

```
let rec get_ptype_dep (p:pType): int
```

Returns depth (here is number of dereference) of a ptype.

```
let rec get_ptype_btype (p:pType): bType
```

Returns btype from ptype.

```
let rec get_btype (s:store)(e:env)(h:heap)(a:aexp): bType
```

Returns btype of a given aexp, when an expression will be evaluated as location this function retrives btype knowing what can generate a location (eg. $@ide \rightarrow type\ of\ ide$).

Declaration:

```
let rec eval_decs (d:dec list)(r:env)(s:store)
    ...
  | Dec(x,Pointer(pType))::decls
            -> let newaddr=(newmem s) (* entry in store *)
            in
            let pdepth=(get_ptype_dep pType)
            in
            let pbtype=(get_ptype_btype pType)
            in
            let r' =
           (updateenv(r,x,Descr_Pointer(Loc(newaddr),pdepth,pbtype)))
            in
            eval_decs decls r' (updatemem(s,Loc(newaddr),ValueLoc(NULL)))
```

Declaration of a pointer, no allocation is done and its first usable location is set to NULL.

Execution

`let rec exec (c: cmd)(r: env)(s: store)(h:heap)`

This function is too long to be reported with code in this report so I'll explain its modification without using too much code.
`exec` with assignment, allocation and deallocation can perform changes on heap so this function *returns heap* along with store.
What has been modified?

- ```
  Ass(i,e) ->
      LVar(id) ->
          | Descr_Pointer(ploc,pdep,ptype) ->
  ```

  (`ploc` = pointer location in store, `pdep` = pointer depth (declaration) and `ptype` = pointer btype)

  Above pattern matching is used when an assignment of the shape `p := <something>` (with p declared as pointer). A pointer is a variable containing a location, when used without dereference, so we can assign a location to it.
  This function perform checks and returns a modified memory containing the assignment just done.

- ```
  Ass(i,e) ->
      LVar(id) ->
          | LPointer(exp) ->
  ```

  (`exp` = a generic pexp)

  This match is for assignment too but is used for command of the shape `^^p := <something>` .
  Here we have on the left a *pexp* and on the right a generic *aexp*, this function must check if this assignment can be done and then it performs some change in memory (store and heap).

- Minor changes to match "ValueLoc" or to take care of heap are done on other commands.

- `| Free(i) ->`

  (`i` = identifier)
  This function is used primarly to remove a reference (that is recursive), because with reference counter when a "cell" ends to have zero reference it's removed from heap and its location is free again.

- `| Alloc(i) ->`

  (`i` = identifier)
  Inside here we allocate memory for pointers, using depth and btype we can know how many cell is needed and what kind of value will be inserted as default in final location.
  Allocation is divided in three main phases:

  - free memory pointed by pointer (if pointer is not null)
  - obtain memory cell "linked" by each other
  - assign and update reference counter of "cells"

4 Difficulties

My first code end nowhere because I poorly written down specification of what to do, so without a clearly idea of how implement features I reach a point where code needed to fix the interpreter was bigger than code to begin a new project.
This isn't really a project problem but a problem for who write code (me in this case), this is the main reason for putting definition of features and chooises of implementation at first in this report.

Real problem:

- Often I found myself stuck thinking about **depth**, as already explained before I used this type of check for telling if an assignment can be done, but sometimes it was tricky to tell why some check fails to perform.
- **debugging**: I found that ocaml compiler is not so verbose in some cases, so I used some *exception* with string output to better understand what is broken.
- **performance**: I found some big performance issues when dealing with dynamic memory. A lot of time was spent trying to fix those problems, but working around on some memory checks and operations improves greatly execution time.
  This is a problem derived from *reference counter* that forces to manipulated memory for every pointer operation.
- **memory leaks**: using pointer inside the main program gives no problem at all because a location is always reachable in some way or if it isn't we already takes care of its deallocation, but when we use *procedures* we must think that what we declared inside will be lost when we returns from *call*.
  Procedure share heap with main program so if we declare and suddenly allocate a pointer (inside procedure) we a an environment reference to a memory location that doesn't exist outside.
  Maybe it's possible to keep trace of what declared inside and wipe it when we exit a procedure but if we do a similar thing we have a "garbage collector" so in my implementation I leave full memory controlo to CC user.

5 Test

I've added some program written in CC that show how new features work (input directory).
Now we'll see some of those test:

test_prj_1.cre:

```
program
 var x:  int;
 var y:  int;
 var p :  ^int;
 var q:  ^^float
begin
 x := 1;
 p := @x;  // now ^p and x share the same value
 y := ^p + 4;
 write(y)
end
```

This program will print:
5

test_prj_2.cre

```
program
 var x:  int;
 var y:  int;
 var p :  ^int;
 var q :  ^int
begin
 alloc(p);  // allocation of p
 x := 5;
 y := 50;
 ^p := x;
 q := @y; // now ^q and y share the same value
 x := x + 5;
 write(^p);
 write(x);
 free(p);  // deallocation of p
 write(y);
 write(^q);
 y := y + 10;  // here changes made to y will be reflected on ^q
 write(q̂)
end
```

This program will print:
5
10
50

50
60

test_prj_3.cre

```
program
 var p :  ^^^int;
 var q :  ^^^int
begin
 alloc(p);
 alloc(q);
 ^^^p := 10;
 ^^^q := 20;
 write(@p);
 write(p);
 write(^p);
 write(^^p);
 write(^^^p);
 write(@q);
 write(q);
 write(^q);
 write(^^q);
 write(^^^q);
 ^^^p := ^^^q; // this assignment can be done
 ^^p := ^q  // but this no
end
```

This program will print;
0
1000
1001
1002
10
1
1003
1004
1005
20
Fatal error: exception Interpreter.PTR_ASSIGN("Different type (depth)")