# Pawns

Pointer Assignment With No Suprises

Leonid Doroszko, Szymon Wojtulewicz

Paradygmaty Jezyków Programowania 2024/25

# Pawns

## Author

Lee Naish

https://lee-naish.github.io/src/pawns/index.html
https://www.mdpi.com/2674-113X/3/4/23
https://peerj.com/articles/cs-22/f

# Agenda

- Motivation
- Overview of the language
- Destructive update
- Sharing analisis
- Other features

- Safety and modularity of pure functions
- Performance and simplicity of (destructively) updating a value under a pointer or reference
- Encapsulation of impurity

# BST Types

```
data BST = Empty | Node BST Int BST -- binary search tree of integers
data List t = Nil | Cons t (List t) -- polymorphic Lists (built in)
type Ints = List Int                -- list of integers
```

## Pure BST insert

```
bst_insert_pure:: BST -> Int -> BST
bst_insert_pure t0 x =
    case t0 of
    Empty ->
        Node Empty x Empty
    (Node l n r) ->
        if x <= n then
            Node (bst_insert_pure l x) n r
        else
            Node l n (bst_insert_pure r x)
```

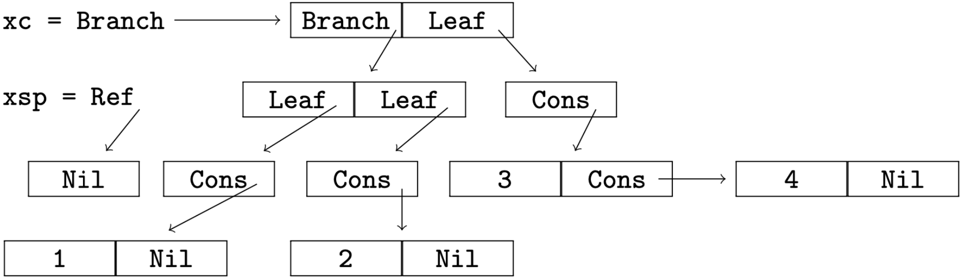
```
-- standard library foldl for lists
foldl:: (b -> a -> b) -> b -> List a -> b
foldl f y xs =
    case xs of
    Nil ->
        y
    (Cons x xs1) ->
        foldl f (f y x) xs1
```

```
list_bst_pure:: Ints -> BST
list_bst_pure xs =
    foldl bst_insert_pure Empty xs
```

# Cord (concrete) example

```
data List t = Nil | Cons t (List t) -- polymorphic Lists (built in)
type Ints = List Int                -- list of integers

data Cord = Leaf Ints | Branch Cord Cord
```



```
x = 42;             -- let binding of x to 42
*xp = x;            -- xp points to a new memory cell containing 42
yp = xp;            -- yp points to the same memory cell (aliases xp)
y = *yp;            -- y is the contents of the memory cell (42)
*!xp := 43 !yp;     -- update what xp points to (also affects yp!)
z = *yp             -- z is the contents of the memory cell (43)
```

## BST insert with pointers

```
bst_insert_pure_p t0 x =
    case t0 of
    Empty ->
        Node Empty x Empty
    (Node *lp *np *rp) ->    -- creates refs/pointers to Node arguments
        if x <= *np then
            Node (bst_insert_pure_p *lp x) *np *rp
        else
            Node *lp *np (bst_insert_pure_p *rp x)
```

```
bst_insert_du tp x =                    -- returns (), *tp updated
    case *tp of
    Empty ->
        *!tp := Node Empty x Empty  -- insert new node, return ()
    (Node *lp n *rp) ->
        if x <= n then
            (bst_insert_du !lp x) !tp -- update lp (and tp!)
        else
            (bst_insert_du !rp x) !tp -- update rp (and tp!)
```

# List to BST with destructive update

```
list_bst_du:: Ints -> BST            -- behaves as a pure function
list_bst_du xs =
    *tp = Empty;                     -- allocate mem cell; init to Empty
    foldl_du bst_insert_du !tp xs -- repeatedly insert element
```

```
foldl_du f y xs =                    -- returns (), y updated
    case xs of
    Nil -> ()                        -- return ()
    (Cons x xs1) ->
        f !y x;                      -- y updated by f
        foldl_du f !y xs1            -- y updated further
```

- 20x performance improvement

## Destructive update with sharing

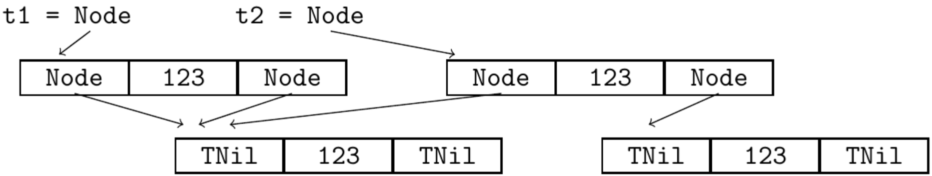
```
sharing_eg:: () -> BST
    subt = Node Empty 42 Empty;
    *tp = Node subt 42 subt;         -- *tp has 3 nodes, subtrees share
    -- bst_insert_pure *tp 42        -- returns BST with 4 nodes
    bst_insert_du !tp 42;            -- inserts 42 into *both* subtrees
    *tp                              -- returns BST with 5 nodes
```

## Possible preconditions

```
assign :: Ref t -> t -> ()
    sharing assign !p v = _
```

- `nosharing`
- `xc = abstract`
- `xc = xs`
- `bp = Node x1 x x2`

When we write `abstract` we assume, that we know nothing about the actual structure of this object, and only know something about its value. For example for two BST nodes `t1` ans `t2`:




We cannot update anything with `abstract`.

1. We want to check some conditions, but basically we want to know which variables share memory
2. We want to overestimate sharing (and be imprecise), because we cannot calculate it exactly
3. For this we want some "abstract domain", or some subset of information (age for employees)
4. This abstract domain consists of pairs of "words" that variables share

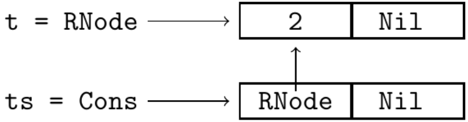
For each variable we want to point to its components in main memory.

```
data RTrees = Nil | Cons RTree RTrees
data RTree = RNode Int RTrees
```

For type `RTrees` we have the components `[]` (this folded path represents both `[Cons.2]` and `[Cons.1,RNode.2]`, since they are of type RTrees), `[Cons.1]` and `[Cons.1,RNode.1]`.
For type `RTree` we have the components `[]` (for `[RNode.2,Cons.1]`, of type RTree), `[RNode.1]` and `[RNode.2]` (which is also the folded version of `[RNode.2,Cons.2]`, of type `RTrees`).

Sharing — set of pairs of these components.

```
t = RNode 2 Nil
ts = Cons t Nil
```




```
{{t.[RNode.1], t.[RNode.1]},
 {t.[RNode.2], t.[RNode.2]},
 {ts.[], ts.[]},
 {ts.[Cons.1], ts.[Cons.1]},
 {ts.[Cons.1,RNode.1], ts.[Cons.1,RNode.1]},
 {t.[RNode.1], ts.[Cons.1,RNode.1]},
 {t.[RNode.2], ts.[]}}
```

1. Transform function to Core Pawns
2. Interpret precondition as set of pairs of components (alias set)
3. For each statement in function modify current alias set
4. Also for each statement using current sharing check for some other conditions
5. Check if end sharing is subset of (precondition $\cup$ postcondition)

We may also have other function calls inside. We do this as induction over depth of function calls.

```
data Stat =                          -- Statement, eg
    Seq Stat Stat |                  -- stat1 ; stat2
    EqVar Var Var |                  -- v = v1
    EqDeref Var Var |                -- v = *v1
    DerefEq Var Var |                -- *v = v1
    DC Var DCons [Var] |             -- v = Cons v1 v2
    Case Var [(Pat, Stat)] |         -- case v of pat1 -> stat1 ...
    Error |                          -- (for uncovered cases)
    App Var Var [Var] |              -- v = f v1 v2
    Assign Var Var |                 -- *!v := v1
    Instype Var Var                  -- v = v1::instance_of_v1_type

data Pat =                           -- patterns for case, eg
    Pat DCons [Var]                  -- (Cons *v1 *v2)
```

# Sharing – all conditions

Compiler assume that the following holds for all depth of function calls less then D, and prove that it also holds for D.

1. for all function calls and assignment statements in $f$ , any live variable that may be updated at that point in an execution of $f$ is annotated with "!"
2. there is no update of live "abstract" variables when executing $f$
3. all parameters of $f$ which may be updated when executing $f$ are declared mutable in the type signature of $f$
4. the union of the pre- and post-conditions of $f$ abstracts the state when $f$ returns plus the values of mutable parameters in all states during the execution of $f$
5. for all function calls and assignment statements in $f$, any live variable that may be directly updated at that point is updated with a value of the same type or a more general type
6. for all function calls and assignment statements in $f$, any live variable that may be indirectly updated at that point only shares with variables of the same type or a more general type

```
alias (Seq stat1 stat2) a0 =                    -- stat1; stat2
   alias stat2 (alias stat1 a0)
alias (EqVar v1 v2) a0 =                        -- v1 = v2
   let
      self1 =  {{v1.c1, v1.c2} | {v2.c1, v2.c2} ∈ a0}
      share1 =  {{v1.c1, v.c2} | {v2.c1, v.c2} ∈ a0}
   in
      a0 ∪ self1 ∪ share1
alias (DerefEq v1 v2) a0 =                      -- *v1 = v2
   let
      self1 =  {{v1.[Ref.1], v1.[Ref.1]}} ∪
                {{fc(v1.(Ref.1 :c1)), fc(v1.(Ref.1 :c2))} | {v2.c1, v2.c2} ∈ a0}
      share1 =  {{fc(v1.(Ref.1 :c1)), v.c2} | {v2.c1, v.c2} ∈ a0}
   in
      a0 ∪ self1 ∪ share1
```

```
bst_sum:: BST -> Int  -- sum of integers in a BST (pure interface)
bst_sum t =
    !init_nsum 0;   -- like nsum = 0
    !bst_sum_sv t;  -- like nsum' = bst_sum_sv t nsum
    *nsum           -- like nsum'

!nsum:: Ref Int  -- declares state variable, nsum

init_nsum:: Int -> ()
    implicit wo nsum  -- binds/initialises/writes nsum
init_nsum n =
    *nsum = n

bst_sum_sv:: BST -> () -- adds all integers in BST to nsum
    implicit rw nsum   -- reads and writes nsum
bst_sum_sv t =
    case t of
    Empty -> ()
    (Node l n r) ->
        *!nsum := *nsum + n;  -- adds n to nsum
        !bst_sum_sv l;        -- adds ints in l (could do same for r)
        *!nsum := *nsum + (bst_sum r) -- uses encapsulated impurity
```

```
put_char: Int -> ()
    implicit rw io
put_char i = as_C "{putchar((int) i);}"

-- pseudo-random number sequence interface
init_random:: int -> () -- initialize sequence with a seed
    implicit wo random_state
random_num:: () -> int  -- return next number in sequence
    implicit rw random_state
```

```
    *xsp = Nil;         -- Nil is a list of any type
    xsp1 = xsp;         -- xsp1 has the same polymorphic type as xsp
    ys = (int_fn !xsp) !xsp1; -- int_fn accepts a ref to a list of ints
    -- Now *xsp (and *xsp1) may be a non-empty list of ints!
    zs = bst_fn *xsp1; -- OOPS! bst_fn accepts a ref to a list of BSTs

cord_list xc =
    *xsp = Nil::Ints;             -- instantiate list type explicitly
    np = cord_list_a !xc !xsp;    -- xsp has a monomorphic type
    *xsp
```

# Pawns

- Higher-order programing
- Formal proof of safety