Sharing analysis in the Pawns compiler

Lee Naish
Computing and Information Systems
University of Melbourne

Pawns: What and why

What: Pawns (tinyurl.com/pawns-lang) is another take on combining declarative and imperative programming

Why: Some things in declarative languages are much slower and more cumbersome than they should be

Pawns supports the typical strict functional programming style but also allows you to get pointers to possibly shared data structures and destructively update them

The language and compiler support expression and analysis of sharing/alias information so that impurity can be encapsulated

Outline

Binary search tree insertion

The efficient, dangerous way: pointers and destructive update

```
void bst_insert_du(long x, tree *tp) {
    while(*tp) {
        if (x <= (*tp)->data)
            tp = &(*tp)->left;
        else
            tp = &(*tp)->right;
    }
    *tp = malloc(sizeof(struct tree_node));
    (*tp)->left = NULL;
    (*tp)->data = x;
    (*tp)->right = NULL;
}
```

Time to insert 30000 elements: 2.22s

Binary search tree insertion

The inefficient, safe way: reconstruct the path down the tree

```
data Bst = Empty | Node Bst Int Bst

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

Time to insert 30000 elements: 51.36s
With `STRef` (destructive update): 4.80s

Binary search tree insertion

Language	DU?	other coding details	time
Pawns	yes		1.10
C	yes		2.20
MLton	yes	uses ref	3.28
Haskell	yes	uses STRef	4.80
MLton	no		7.44
MLton	no	uses ref	10.70
C	no	iterative, GC_MALLOC, no free	15.44
Pawns	no		16.25
Haskell	no	uses ‘seq‘ for strictness	21.75
C	no	iterative, malloc, free	21.85
C	no	iterative, GC_MALLOC, GC_FREE	22.13
C	no	recursive, malloc, free	28.61
Haskell	no	no ‘seq‘	51.36

Pawns binary search tree insertion

```
type bst ---> empty ; node(bst, int, bst).

bst_insert_du :: int -> ref(bst) -> void
    sharing bst_insert_du(x, !tp) = v
    pre nosharing     post nosharing.
bst_insert_du(x, tp) = {
    cases *tp of {
    case node(*lp, n, *rp):
        if x <= n then
            bst_insert_du(x, !lp) !tp
        else
            bst_insert_du(x, !rp) !tp
    case empty:
        *!tp := node(empty, x, empty)
}   }.
```

Pawns binary search tree building

```
list_bst :: list(int) -> bst.
list_bst(xs) = {
    *tp = empty;
    list_bst_du(xs, !tp);
    *tp }.

list_bst_du :: list(int) -> ref(bst) -> void
    sharing list_bst_du(xs, !tp) = v
    pre xs = abstract     post nosharing.
list_bst_du(xs, tp) = {
    cases xs of {
    case cons(x, xs1):
        bst_insert_du(x, !tp);
        list_bst_du(xs1, !tp)
    case nil: void    }}.
```

Summary of Pawns features

Functional programming with algebraic data types, refs/pointers

Pointers to arguments of data constructors can be obtained by pattern matching

Pointers to values can be obtained, but not pointers to variables

Assignment via pointers; mutability of function arguments declared; live variables annotated where they may be updated

Pawns = Pointer Assignment Without Nasty Surprises

Sharing declared in pre- and post-conditions of functions; can share with "abstract" (unknown/any sharing, update not allowed)

Not covered here: "state variables" (like global variables but impurity also encapsulated)

Core Pawns

An early pass of the compiler eliminates nested expressions etc

```
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)] | -- cases 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]               -- case cons(*v1, *v2)
```

Sharing (and type) analysis: the aim

For all functions *f*, if the precondition of *f* is always satisfied

1. for all function calls and assignment statements in *f*, any live variable that may be updated at that point 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 return state plus the values of mutable parameters in all intermediate states,
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, and
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.

Abstract interpretation domain

We abstractly interpret each function, starting with the precondition

The abstract domain is a set of pairs of variable *components* which may share, including “self sharing”

Variable components are paths from the top level of a value to the argument of a data constructor; recursive types are “folded” (function `fc`) to get a finite number of components

```
type maybe(T) ---> just(T); nothing.
type either(A, B) ---> left(A); right(B).
type list(T) ---> cons(T, list(T)); nil.
```

`x` of type `maybe(either(bool, int))` has components `x.[just.1]`, `x.[just.1,left.1]` and `x.[just.1,right.1]`

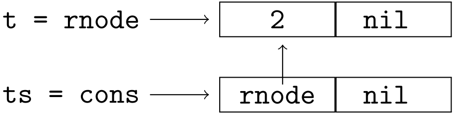
`ys` of type `list(int)` has components `ys.[cons.1]` and `ys.[]`

Abstract domain example

```
type rtrees = list(rtree).
type rtree ---> rnode(int, rtrees).
```

rtrees components: `[]`, `[cons.1]` and `[cons.1,rnode.1]`
rtree components: `[]`, `[rnode.1]` and `[rnode.2]`

```
t = rnode(2, nil);
ts = cons(t, nil)
```



t = rnode → 2 | nil

ts = cons → rnode | 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.[]}}
```

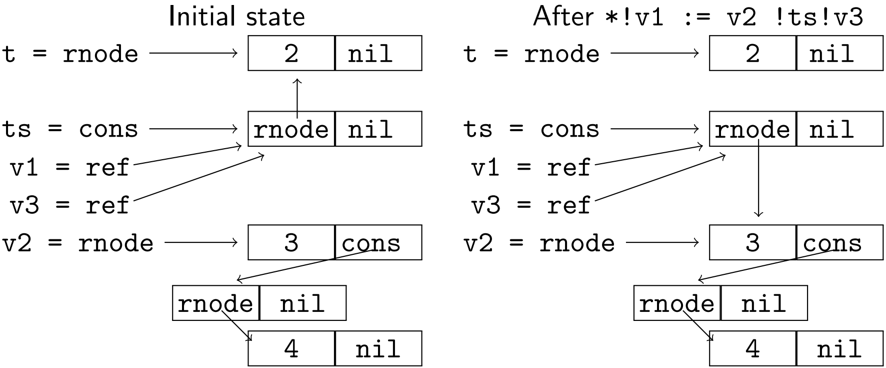
Abstract interpretation of Seq, EqVar, DerefEq

```
alias (Seq stat1 stat2) a0 =                -- stat1; stat2
   alias stat2 (alias stat1 a0)
alias (EqVar v1 v2) a0 =                    -- v1 = v2
   let
      self1 =  $\{\{\mathtt{v1}.c_1, \mathtt{v1}.c_2\} \mid \{\mathtt{v2}.c_1, \mathtt{v2}.c_2\} \in \mathtt{a0}\}$
      share1 =  $\{\{\mathtt{v1}.c_1, v.c_2\} \mid \{\mathtt{v2}.c_1, v.c_2\} \in \mathtt{a0}\}$
   in
      $\mathtt{a0} \cup \mathtt{self1} \cup \mathtt{share1}$
alias (DerefEq v1 v2) a0 =                  -- *v1 = v2
   let
      self1 =  $\{\{\mathtt{v1}.[\mathtt{ref}.1], \mathtt{v1}.[\mathtt{ref}.1]\}\} \cup$
               $\{\{\mathtt{fc}(\mathtt{v1}.(\mathtt{ref}.1 : c_1)), \mathtt{fc}(\mathtt{v1}.(\mathtt{ref}.1 : c_2))\} \mid \{\mathtt{v2}.c_1, \mathtt{v2}.c_2\}$
      share1 =  $\{\{\mathtt{fc}(\mathtt{v1}.(\mathtt{ref}.1 : c_1)), v.c_2\} \mid \{\mathtt{v2}.c_1, v.c_2\} \in \mathtt{a0}\}$
   in
      $\mathtt{a0} \cup \mathtt{self1} \cup \mathtt{share1}$
```

Abstract interpretation of Assign

```
alias (Assign v1 v2) a0 =                        -- *v1 := v2
   let
      al =  $\{v_a.c_a \mid \{\texttt{v1}.[\texttt{ref.1}], v_a.c_a\} \in \texttt{a0}\}$
      -- (check annotations, sharing with abstract)
      self1al =  $\{\{\texttt{fc}(v_a.(c_a{+}{+}c_1)), \texttt{fc}(v_b.(c_b{+}{+}c_2))\} \mid$
                 $v_a.c_a \in \texttt{al} \wedge v_b.c_b \in \texttt{al} \wedge \{\texttt{v2}.c_1, \texttt{v2}.c_2\} \in \texttt{a0}\}$
      share1al =  $\{\{\texttt{fc}(v_a.(c_a{+}{+}c_1)), v.c_2\} \mid$
                 $v_a.c_a \in \texttt{al} \wedge \{\texttt{v2}.c_1, v.c_2\} \in \texttt{a0}\}$
   in if v1 is a mutable parameter then
         $\texttt{a0} \cup \texttt{self1al} \cup \texttt{share1al}$
      else let
         -- old1 = old aliases for v1, which can be removed
         old1 =  $\{\{\texttt{v1}.(\texttt{ref.1} : d : c_1), v.c_2\} \mid$
                 $\{\texttt{v1}.(\texttt{ref.1} : d : c_1), v.c_2\} \in \texttt{a0}\}$
      in $(\texttt{a0} \setminus \texttt{old1}) \cup \texttt{self1al} \cup \texttt{share1al}$
```

Assign example 1



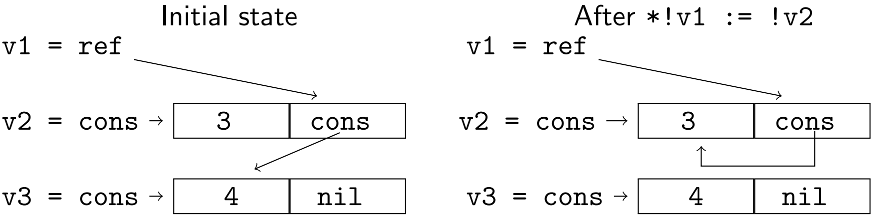
Initial state

After `*!v1 := v2 !ts!v3`

`ts`, `v1`, `v3` and `v2` sharing added

`v1` and `t` sharing removed

Assign example 2



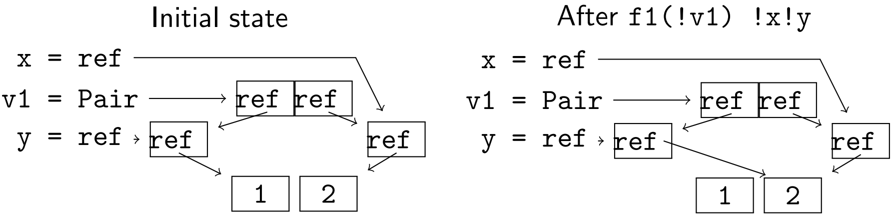
v1 and v3 sharing removed then added again

Abstract interpretation of App (ignoring closures)

```
alias (App v f [$v_1, \ldots v_N$]) a0 =          -- v = f(v1...vN)
   let
      -- (check renamed precondition and annotations)
      mut = the arguments that are declared mutable
      post = renamed postcondition + precondition for mut
      pt =  $\{\{x_1.c_1, x_3.c_3\} \mid \{x_1.c_1, x_2.c_2\} \in \texttt{post} \wedge \{x_2.c_2, x_3.c_3\} \in \texttt{a0}\}$
      pm =  $\{\{x_1.c_1, x_2.c_2\} \mid \{x_1.c_1, v_i.c_3\} \in \texttt{a0} \wedge \{x_2.c_2, v_j.c_4\} \in \texttt{a0} \wedge$
               $\{v_i.c_3, v_j.c_4\} \in \texttt{post} \wedge v_i \in \texttt{mut} \wedge v_j \in \texttt{mut}\}$
   in
      $\texttt{a0} \cup \texttt{pt} \cup \texttt{pm}$
```

Note: the precondition for non-mutable arguments is not added

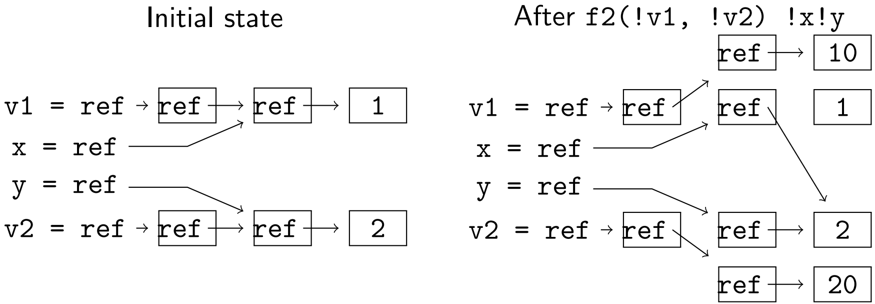
App example 1



Mutable argument components are proxies for everything they share with

```
f1 :: pair(ref(ref(int)), ref(ref(int))) -> void
    sharing f1(!v1) = r
    pre nosharing      post *a = *b; v1 = pair(a, b).
f1(v1) =
    cases v1 of {case pair(rr1, rr2): *rr1 := *rr2 !v1}.
```

App example 2



f2 can be written so that v1 and v2 never share during the execution

```
f2 :: ref(ref(ref(int))) -> ref(ref(ref(int))) -> void
    sharing f2(!v1, !v2) = v pre nosharing post **v1 = **v2.
f2(v1, v2) = {*r10 = 10; *rr10 = r10; *r20 = 20; *rr20 = r20;
    rr1 = *v1; rr2 = *v2; *!v1 := rr10; *!v2 := rr20;
    *rr1 := *rr2 !v1!v2}.
```

Abstract interpretation of other cases

Function applications can result in closures that contain data structures which can be shared and updated

Case statements can remove some sharing for each branch but lose some precision due to the possibility of cyclic structures

See the paper for details

Implementation status

Implementation in Prolog, standard set library used (binary search trees), no work done on optimisation

Speed seems fine, though no stress testing done - analysis and translation of Pawns to C is faster than compilation of C

Various bugs discovered when the paper was written; not yet fixed

Conclusion

Destructive update via pointers to possibly shared data is efficient but hard to incorporate nicely into declarative languages

You can have destructive update of algebraic data types without adding explicit refs or similar to the data type

Such destructive update can be encapsulated inside a pure interface

The main cost (and also benefit) in Pawns is extra declarations and annotations concerning sharing and mutability in the code

The extra analysis in the compiler is complicated but seems to be possible with acceptable efficiency