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

Motivation

Pawns features

Core Pawns

Sharing analysis overview

Sharing analysis abstract domain

Sharing analysis algorithm

Conclusion
Binary search tree insertion

The efficient, dangerous way: pointers and destructive update

```c
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

<table>
<thead>
<tr>
<th>Language</th>
<th>DU?</th>
<th>other coding details</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pawns</td>
<td>yes</td>
<td></td>
<td>1.10</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
<td></td>
<td>2.20</td>
</tr>
<tr>
<td>MLton</td>
<td>yes</td>
<td>uses ref</td>
<td>3.28</td>
</tr>
<tr>
<td>Haskell</td>
<td>yes</td>
<td>uses STRef</td>
<td>4.80</td>
</tr>
<tr>
<td>MLton</td>
<td>no</td>
<td></td>
<td>7.44</td>
</tr>
<tr>
<td>MLton</td>
<td>no</td>
<td>uses ref</td>
<td>10.70</td>
</tr>
<tr>
<td>C</td>
<td>no</td>
<td>iterative, GC_MALLOC, no free</td>
<td>15.44</td>
</tr>
<tr>
<td>Pawns</td>
<td>no</td>
<td></td>
<td>16.25</td>
</tr>
<tr>
<td>Haskell</td>
<td>no</td>
<td>uses ‘seq’ for strictness</td>
<td>21.75</td>
</tr>
<tr>
<td>C</td>
<td>no</td>
<td>iterative, malloc, free</td>
<td>21.85</td>
</tr>
<tr>
<td>C</td>
<td>no</td>
<td>iterative, GC_MALLOC, GC_FREE</td>
<td>22.13</td>
</tr>
<tr>
<td>Haskell</td>
<td>no</td>
<td>recursive, malloc, free</td>
<td>28.61</td>
</tr>
<tr>
<td></td>
<td></td>
<td>no ‘seq’</td>
<td>51.36</td>
</tr>
</tbody>
</table>
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)
} }. 

Naish, Lee (Melbourne Uni.) Sharing analysis in the Pawns compiler October 18, 2015 7 / 23
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)
An early pass of the compiler eliminates nested expressions etc

```haskell
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)
```

Core Pawns

Naish, Lee (Melbourne Uni.) Sharing analysis in the Pawns compiler October 18, 2015 10 / 23
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

\[
\text{type } \text{maybe}(T) \rightarrow \text{just}(T); \text{nothing}.
\]
\[
\text{type } \text{either}(A, B) \rightarrow \text{left}(A); \text{right}(B).
\]
\[
\text{type } \text{list}(T) \rightarrow \text{cons}(T, \text{list}(T)); \text{nil}.
\]

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

\( ys \) of type \( \text{list}(\text{int}) \) has components \( ys.[\text{cons.1}] \) and \( ys.[\text{}] \)
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]

\[
\begin{align*}
t &= \text{rnode}(2, \text{nil}); \\
\text{ts} &= \text{cons}(t, \text{nil})
\end{align*}
\]

\[
\begin{align*}
t &= \text{rnode} \quad \rightarrow \quad & \begin{array}{c}
2 \\
\downarrow \\
\text{nil}
\end{array} \\
\text{ts} &= \text{cons} \quad \rightarrow \quad & \begin{array}{c}
\text{rnode} \\
\text{nil}
\end{array}
\end{align*}
\]

\{
{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 =  {{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}
      share1 =  {{fc(v1.(ref.1 :c1)), v.c2} | {v2.c1, v}.c2} ∈ a0}
in
      a0 ∪ self1 ∪ share1
Abstract interpretation of Assign

alias (Assign v1 v2) a0 =          -- *v1 := v2
  let
    al = \{νa.c_a \mid \{v1.[ref.1], νa.c_a\} ∈ a0\}
    -- (check annotations, sharing with abstract)
    self1al = \{\{fc(νa.(c_a++c_1)),fc(νb.(c_b++c_2))\} \mid 
                  νa.c_a ∈ al ∧ νb.c_b ∈ al ∧ \{v2.c_1, v2.c_2\} ∈ a0\}
    share1al = \{\{fc(νa.(c_a++c_1)), ν.c_2\} \mid 
                  νa.c_a ∈ al ∧ \{v2.c_1, ν.c_2\} ∈ a0\}
  in if v1 is a mutable parameter then
      a0 ∪ self1al ∪ share1al
  else let
    -- old1 = old aliases for v1, which can be removed
    old1 = \{\{v1.(ref.1:d : c_1), ν.c_2\} \mid 
                 \{v1.(ref.1:d : c_1), ν.c_2\} ∈ a0\}
    in (a0 \ old1) ∪ self1al ∪ share1al
Assign example 1

**Initial state**

- \( t = \text{rnode} \) → 2 nil
- \( \text{ts} = \text{cons} \) → \( \text{rnode} \) nil
- \( v1 = \text{ref} \)
- \( v3 = \text{ref} \)
- \( v2 = \text{rnode} \) → 3 cons

- \( \text{rnode} \) nil
- 4 nil

**After \(*!v1 := v2 \ !ts!v3*\)**

- \( t = \text{rnode} \) → 2 nil
- \( \text{ts} = \text{cons} \) → \( \text{rnode} \) nil
- \( v1 = \text{ref} \)
- \( v3 = \text{ref} \)
- \( v2 = \text{rnode} \) → 3 cons

- \( \text{rnode} \) nil
- 4 nil

- \( \text{ts}, v1, v3 \) and \( v2 \) sharing added
- \( v1 \) and \( t \) sharing removed
Assign example 2

Initial state

\[ v_1 = \text{ref} \]
\[ v_2 = \text{cons} \rightarrow \begin{array}{c} 3 \\ \text{cons} \end{array} \]
\[ v_3 = \text{cons} \rightarrow \begin{array}{c} 4 \\ \text{nil} \end{array} \]

After \(!v_1 := !v_2\)

\[ v_1 = \text{ref} \]
\[ v_2 = \text{cons} \rightarrow \begin{array}{c} 3 \\ \text{cons} \end{array} \]
\[ v_3 = \text{cons} \rightarrow \begin{array}{c} 4 \\ \text{nil} \end{array} \]

\(v_1\) and \(v_3\) sharing removed then added again
Abstract interpretation of App (ignoring closures)

alias (App v f [v₁,...vₙ]) 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 = { {x₁.c₁, x₃.c₃} | {x₁.c₁, x₂.c₂} ∈ post ∧ {x₂.c₂, x₃.c₃} ∈ a₀ }
pm = { {x₁.c₁, x₂.c₂} | {x₁.c₁, vᵢ.c₃} ∈ a₀ ∧ {x₂.c₂, vⱼ.c₄} ∈ a₀ ∧ 
  {vᵢ.c₃, vⱼ.c₄} ∈ post ∧ vᵢ ∈ mut ∧ vⱼ ∈ mut }
in
  a₀ ∪ pt ∪ pm

Note: the precondition for non-mutable arguments is not added
Mutable argument components are proxies for everything they share with

\[ f_1 :: \text{pair}(	ext{ref}(	ext{ref}(	ext{int})), \text{ref}(	ext{ref}(	ext{int}))) \rightarrow \text{void} \]

\[ \text{sharing } f_1(!v1) = r \]

\[ \text{pre nosharing } \text{post } *a = *b; v1 = \text{pair}(a, b). \]

\[ f_1(v1) = \]

\[ \text{cases } v1 \text{ of } \{ \text{case pair}(rr1, rr2): *rr1 := *rr2 !v1 \}. \]
App example 2

Initial state

\begin{align*}
\text{v1} &= \text{ref} \rightarrow \text{ref} \rightarrow \text{ref} \rightarrow 1 \\
\text{x} &= \text{ref} \\
\text{y} &= \text{ref} \\
\text{v2} &= \text{ref} \rightarrow \text{ref} \rightarrow \text{ref} \rightarrow 2
\end{align*}

After \( f_2(\text{!v1}, \text{!v2}) \) \( !x!y \)

\begin{align*}
\text{v1} &= \text{ref} \rightarrow \text{ref} \rightarrow \text{ref} \rightarrow 1 \\
\text{x} &= \text{ref} \\
\text{y} &= \text{ref} \\
\text{v2} &= \text{ref} \rightarrow \text{ref} \rightarrow \text{ref} \rightarrow 2
\end{align*}

\( f_2 \) can be written so that \( v_1 \) and \( v_2 \) never share during the execution

\[ f_2 :: \text{ref(ref(ref(int))))} \rightarrow \text{ref(ref(ref(int))))} \rightarrow \text{void} \]

\[ \text{sharing } f_2(\text{!v1}, \text{!v2}) = v \text{ pre nosharing post } **v_1 = **v_2. \]

\[ f_2(v_1, v_2) = \{ *r_10 = 10; *rr_10 = r_{10}; *r_20 = 20; *rr_20 = r_{20}; \]
\[ rr_1 = *v_1; rr_2 = *v_2; !v_1 := rr_{10}; !v_2 := rr_{20}; \]
\[ *rr_1 := *rr_2 \text{ !v1!v2}. \]
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.