Pawns: a declarative/imperative language

Lee Naish
Computing and Information Systems
University of Melbourne

tinyurl.com/pawns-lang

Outline

Why bother?

Adding pointers and destructive update to FP

Pointers and destructive update in Pawns

Sharing analysis

Encapsulating impurity

IO and state variables

Implementation

Conclusion

Why bother?

FP is great: algebraic data types, polymorphism, higher order, no "nasty surprises" when evaluating a function, eg `foo x y`

But sometimes destructive update of shared structures via pointers is much more efficient or simple to express than the alternatives

Eg, graph reduction, union-find, representing a virtual world containing multiple agents, building a binary search tree, ...

Computer hardware should be used, not simulated in software for the sake of semantic purity

Adding pointers and destructive update to FP

Pointers/references/refs can be represented using a data constructor

```
data Ref t = Ref t  -- pointer to value of type t
```

Then you support the equivalent of `*x = y` in C (may affect `x`, `y` and other vars)

```
-- Haskell
do  -- ST monad
x <- newSTRef w
y <- ...
z <- ...
writeSTRef x y
```

```
(* SML *)

let val x = ref w in
let val y = ...   in
let val z = ...   in
        x := y
```

Distinguishing pure and impure code

The distinction between pure and impure code is done at the type level

Anything without refs cannot be destructively updated and is pure

Anything with refs can be destructively updated by *anything* that has access to it (in Haskell it must be inside the `ST` monad)

You can end up with multiple versions of data types depending on what parts of the data structure you want to update (eg, list element, list tail, both or neither; potentially 256 versions of `zip`)

Converting from a pure data structure to an impure one is particularly painful in Haskell because you must use a monad for the latter

Refs also introduce extra indirection in data structures

The Disciple language does things a bit differently but has a very complex type system

Pointers and destructive update in Pawns

In Pawns you don't have to put refs into data types (no extra indirection or multiple versions, same with Disciple)

Refs/pointers to arguments of data constructors can be created using an extension to pattern matching

All impure code has "!" annotations, checked by the compiler

Its clear what variables can get updated at each point

Pawns = Pointer Assignment With No Surprises

Impurity can be encapsulated: you can create a data structure using destructive update then pass it to pure code

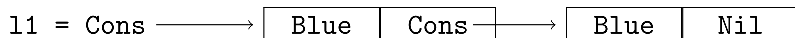
Memory model

The declarative view: variables denote values; memory is irrelevant

```
-- Haskell-ish syntax used for now
data Colour = Red | Green | Blue
data List = Nil | Cons Colour List
l1 = Cons Blue (Cons Blue Nil)
```

The imperative view (needed to understand and use destructive update): some values are represented using words in main memory

Constants are atomic (small integer/one word), data constructors with $N > 0$ arguments are tagged pointers to a block of N main memory words



Pawns has no explicit `Ref` but `*xp` denotes what `xp` points to (like C)

(De)constructing and updating values in Pawns

High level method of replacing the head of a list cs0 with Red:

```
case cs0 of (Cons c cs) -> Cons Red cs
```

In Pawns we can bind pointer variables using “*” in patterns

```
case cs0 of (Cons *cp *csp) -> Cons Red *csp
```

Destructive update is done via pointers (see later for refinement!):

```
case cs0 of (Cons *cp *csp) -> *cp := Red; cs0
```

LHS of let bindings can also have “*” (allocates ptr in memory word):

```
**cs1pp = cs0   -- ptr to ptr to contents/value of cs0
```

Destructive updates are obvious in Pawns

Whenever a variable may be updated the code must have a warning and type signatures must indicate if an argument may be updated

```
*cp := Red -- ERROR   ->    *!cp := Red !cs0 -- OK
```

Building a BST from a list of ints (type signatures refined later):

```
data Tree = Empty |
     Node Tree Int Tree

list_bst:: Ints -> Tree
list_bst xs =
    *tp = Empty
    list_bst_du xs !tp
    *tp
```

```
list_bst_du:: Ints -> !Ref Tree->()
list_bst_du xs !tp =
    case xs of
    (Cons x xs1) ->
        bst_insert_du x !tp
        list_bst_du xs1 !tp
        ()
    Nil -> ()
```

Destructive updates are obvious (cont.)

The function that does the real work:

```
bst_insert_du:: Int -> !Ref Tree -> ()
bst_insert_du x !tp =
    case *tp of
    Empty ->
        *!tp := Node Empty x Empty  -- insert new node
    (Node *lp n *rp) ->
        if x <= n then
            (bst_insert_du x !lp) !tp
        else
            (bst_insert_du x !rp) !tp
```

More elegant than typical imperative versions and more efficient (and slightly shorter) than typical declarative versions

Sharing analysis

In order to write/understand/analyse code with destructive update via pointers, and know what variables and arguments need annotations, you need to understand sharing of data structures

In Pawns you must declare sharing of arguments and results of functions via preconditions and postconditions in the type signature

For BST building/insertion there is no sharing so its trivial, eg:

```
list_bst_du:: Ints -> !Ref Tree -> ()
    sharing list_bst_du xs tp = _
    pre nosharing
    post nosharing
```

But if we have a function that has multiple arguments that are lists that may be updated (for example), we have to think carefully

Sharing (cont.)

Suppose we want a version of ++ which destructively updates (the end of) the first argument and returns the list as well

```
app_du:: !List -> List -> List
    sharing app_du xs ys = zs
    pre nosharing
    post xs = ys; zs = xs
app_du xs ys =
    *xsp = xs
    adu1 !xsp ys
    *xsp

... cs1 = Cons Red Nil
    cs2 = app_du !cs1 (Cons Blue Nil)
    cs3 = app_du !cs1 cs2 --- ERROR*2
```

```
adu1:: !Ref List -> List ->()
    sharing adu1 xsp ys = _
    pre nosharing
    post ys = *xsp
adu1 !xsp ys =
    case *xsp of
    Nil ->
        *!xsp := ys
    (Cons _ *xsp1) ->
        adu1 xsp1 ys
```

Encapsulating impurity

We don't want to think of how all values are represented all the time

By declaring "abstract" sharing it means we don't know or care what a variable shares with and (therefore) it should not be updated

This is the default but we can make it explicit as follows. Thus `list_bst` has a "pure" interface but the implementation uses destructive update!

```
list_bst:: Ints -> Tree
    sharing list_bst xs = t
    pre xs = abstract
    post t = abstract
```

The compiler checks consistency of sharing declarations and annotations, update of "abstract" variables and type preservation

IO and state variables

Pawns programs can declare "state variables"; `io` is built-in

State variables can be declared as implicit arguments and/or results of functions in the type signature: read-write, read-only or write-only

```
get_char :: () -> Int
    implicit rw io

!counter :: Ref Int
use_counter :: List -> List implicit rw counter
init_counter :: Int -> () implicit wo counter
encapsulate_counter :: List -> List  --- pure interface
encapsulate_counter xs = ...
    !init_counter 42              -- binds *counter to 42
    ys = !use_counter xs          -- uses/updates *counter
    ...
```

State variables — pros and cons

State variables are “tame” global variables

- \+ Very easy to add extra state etc
- \+ Argument order etc not an issue
- \+ Diagnostic writes by ignoring “`io` undefined” errors
- +/− Programmers think about global state
- − Need for `wo` functions; could support eg, `!*counter = 0`
- − Code less flexible (based on state variable names, not just their types), including higher order
- − Type checking for higher order a bit more complicated
- − Semantics not quite so clear

Implementation

Pawns is compiled to C with macros etc to support algebraic data types

Very simple translation but `gcc -O3` does a fantastic job

Few builtins/libraries but very easy interface to C (can even have C code to define Pawns functions)

Compiler written in Prolog; uses Prolog operator declarations to support (not so nice) syntax without writing a parser

Some known bugs in sharing analysis implementation, amongst other things, but (apparently) correct algorithm has been published

On github so you are welcome to play around with it (or even re-implement it:-)

Some real Pawns code

```
type bst ---> mt ; node(bst, int, bst).
type ints = list(int).
type rbst = ref(bst).

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

Implementation (cont.)

```
void bst_insert_du(intptr_t x, bst* tp) {
    bst V0 = *tp;
    switch_bst(V0)
    case_mt_ptr()
        bst V2 = mt(); bst V3 = mt(); bst V1 = node(V2, x, V3);
        *tp=V1;
    case_node_ptr(lp, V4, rp)
        intptr_t n = *V4;
        PAWNS_bool V5 = leq(x, n);
        switch_PAWNS_bool(V5)
        case_PAWNS_true_ptr()
            bst_insert_du(x, lp); return;
        case_PAWNS_false_ptr()
            bst_insert_du(x, rp); return;
        end_switch()
    end_switch()
}
```

Conclusion

For most code we want to ignore details of how values are represented

But sometimes we want destructive update of shared structures

Its not easy to reconcile these two views

Sharing analysis can find bounds on the effects of destructive update

Pawns uses a combination of annotations (concerning mutability), pre-conditions and post-conditions (concerning sharing)

This clarifies the appropriate level of abstraction, documents important invariants and allows many potential errors to be detected by the compiler

Binary search tree insertion benchmark

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

What sharing and type analysis proves

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.