Pawns: a declarative/imperative language

Lee Naish
Computing and Information Systems
University of Melbourne
(actually, not quite...)

Contribution (intended at least)

Another take on combining declarative and imperative programming

The general aim is to encourage a declarative style of programming but also allow the use of powerful and "dangerous" imperative constructs (eg, where they can greatly enhance efficiency)

The challenge is to encapsulate the dangerous imperative code so any "contamination" is limited

- Most of the time you can wear board shorts and relax in the tropical paradise that is declarative programming
- There are clear signs when you have to put on a bomb suit and concentrate on avoiding the "surprises" of imperative programming

More specifically...

Typical declarative programming features (but not lazy evaluation)

Typical algebraic data type definitions, but an extension to pattern matching allows access to pointers/references (to arguments of data constructors)

Destructive update (via pointers) whenever you want it

"Interface integrity" via explicit annotation of which variables may be updated at each program point (the implementation does sharing analysis)

Preconditions and postconditions concerning sharing (eg, you can say "whenever this function is called, these two arguments must not share")

Caveat: a work in progress ...

Outline

Declarative and imperative programming

Three views of algebraic data types

(De)constructing and updating values in Pawns

Examples

Other issues

Related work

Conclusion

Declarative and imperative programming

Declarative programming is about manipulating *values*, independently of how they are represented, stored etc. Variables are just names for values

Imperative programming is about storing values in memory locations. Variables are primarily names for memory locations (lvalues)

Destructive update of atomic values is not a big issue — variables can be renamed statically ("x" has three different meanings below):

```
int x,y,z;       x = 42; y = x;   x = x+1;  z = x; ...   ->
int x0,x1,y,z; x0 = 42; y = x0; x1 = x0+1; z = x1; ...
```

Destructive update of non-atomic values *is* a big issue — other values/variables may be affected also, due to sharing of representations

```
list x,y; x = init(); y = x; update(x);
```

Three views of algebraic data types

1) High level view:

```
data Colour = Red | Green | Blue
data List = Nil | Cons Color List
l1 = Cons Blue (Cons Blue Nil)
```

2) "Ref" view: wrap each data constructor argument with a Ref

```
data Colour = Red | Green | Blue
data List = Nil | Cons (Ref Color) (Ref List)
data Ref t = Ref t
l1 = Cons (Ref Blue) (Ref (Cons (Ref Blue) (Ref Nil)))
```

Equivalent to (or a special case of) the high level view

Three views of algebraic data types (cont.)

For each cons cell there are two references/addresses (on the heap)

3) "Store" view: each ref is an integer and there is a separate store

```
data List = Nil | Cons (Ref Color) (Ref List)
data Ref t = Ref Int
l1 = Cons (Ref 65536) (Ref 65544)
l1t = Cons (Ref 70000) (Ref 70008)
store = {65536->Blue, 65544->l1t, 70000->Blue, 70008->Nil}
```

If a single value in the store is changed, several high level values may change

This view is needed to understand destructive update

Why would we ever want the store view?

Updating shared structures is vital to many important efficient algorithms

Eg, converting expressions to a normal form (outermost evaluation with sharing of common sub-expressions):

```
data Expr = Times Expr Expr | Plus Expr Expr | ...
e1 = Times zero x    -- avoid evaluating x
e2 = Times ten x     -- avoid evaluating x ten times
```

Eg, unification in Prolog (if X is bound to Y and Y is bound to Z, then when Z gets bound, so do X and Y):

```
data Term =
    Var Term | -- variables are pointers (possibly to self)
    Nonvar Constructor [Term] -- could refine this
```

(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 expose the ref/pointer view by putting "*" before variables (`*csp` is a list, `csp` is a pointer to a list, like `list *csp` in C):

```
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
```

You can also define pointers to pointers etc (generally requires mallocs):

```
cs1 = Cons Red cs; ***cs1ppp = Cons Red cs -- no Ref or &
```

Destructive updates are obvious in Pawns!

Whenever a variable could be updated, the code must have a warning!
Pawns = Pointer Assignment With No Surprises

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

Example: building a BST of Items from a List

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

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

Examples (cont.)

The function that does the real work:

```
bst_insert_du:: Item -> !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

Examples (cont.)

A cord (containing lists) allows constant time concatenation; we can potentially convert it to a single list in $O(N)$ time and constant space by destructively concatenating the lists it contains

```
data Cord = Leaf List | Branch Cord Cord
...
cordlist1 :: !Cord -> !Ref List -> Ref List
-- csp points to Nil of list we smash; returns ptr to new Nil
cordlist1 !cc !csp =
    case cc of
    (Leaf cs) ->
        *!csp := cs !cc!cs      -- cc, cs are safe
        lastp csp               -- return ptr to Nil of csp
    (Branch cc1 cc2) ->
        csp1 = (cordlist1 !cc1 !csp) !cc!cc2 -- cc2 is safe
        (cordlist1 !cc2 !csp1) !cc!csp
```

Examples (cont.)

How can a programmer reason that `cc`, `cs` and `cc2` are not really modified by the code of `cordlist1`?

Actually, they *can* be modified if the lists in the cord share; the compiler analysis (hopefully) alerts the programmer to this fact

The postcondition of `cordlist1` is the data in the leaves of `cc` share with `csp` and the result (similarly for the precondition)

Functions which add a new list to a cord (eg, `cord_app`) can have a precondition which prevents sharing — a safe interface

```
cord_app:: (cc0::Cord -> cs::List -> cc::Cord)
    precond nosharing
    postcond cc = Branch cc0 (Leaf cs)
...
 cc0 = list_cord cs
 cc1 = cord_app cc0 cs -- ERROR: cc0 and cs share
```

Other issues

Support for

- "Unknown" and "const" sharing
- Arrays (with refs)
- IO, etc
- Pointer equality
- While loops...
- Specifying which parts of a data structure can (not) be updated

Combining polymorphism, higher order and destructive update...

Formal type system, preservation, progress, referential transparency for code without "!"...

Some related work

ML (etc): types can contain explicit updatable refences but not implicit ones (no low level view of high level types). It can be less clear what variables are changed by an update

Disciple: "region" information (like transitive sharing) is declared and/or inferred. It can be less clear what variables are changed by an update. Higher order etc supported (complicated)

Mars: declarative semantics for update — copying is done if sharing analysis concludes that destructive update is not safe (can't destructively update possibly shared things)

Mercury (etc): sharing analysis used for compile time garbage collection and structure re-use (can't destructively update possibly shared things)

Conclusions

Some important algorithms cannot easily be expressed in a declarative way

Algebraic data types can be viewed at different levels of abstraction, allowing the "ref" view without changing the definition/implementation

Sharing analysis seems to provide a reasonable way to find bounds on the effects of destructive update

Explicitly annotating expressions to show what variables may be affected allows us to choose an appropriate level of abstraction when viewing code

A combination of annotations, pre-conditions and post-conditions allows flexibility while alerting programmers to many potential errors

Some real code

Currently syntax is Prolog with operators declared, type processing is very limited, general expressions are not supported in some places, ...

```
func lastp(ref(bs):: q) = ref(bs):: r
    precond nosharing
    postcond ref(bs):: r = q.
lastp(q):
    ref(bs):: q1 = *q;
    cases bs:: q1 of (
    case nil:
        return(ref(bs):: q)
    case cons(*h, *p):
        return(ref(bs):: lastp(p))
    ).
```

Some real code (reformatted)

Compilation to C with macros derived from algebraic data types

```
bs* lastp(bs* q) {
    bs* q1;
    q1 = *q;
    switch_bs(q1)
    case_nil()
        return q;
    case_cons_ptr(h, p)
        return lastp(p);
    end_switch();
}
```

Or, if you prefer Haskell...

```
lastp :: (STRef v (Pbs v)) -> ST v (STRef v (Pbs v))
lastp q = do
 let v1852  = q
 q1   <- readSTRef v1852
 case q1  of
   (Pnil ) -> do
     let v12672  = q
     return v12672
   (Pcons h p ) -> do
     v43803  <- (lastp p )
     return v43803
```