Taming global variables in Pawns

Lee Naish
Computing and Information Systems
University of Melbourne

tinyurl.com/pawns-lang

Have you ever wanted to...

Modify a set of functions/procedures to

- pass in an extra value (eg, a flag),
- pass in and return extra values (eg, current and final values of a counter),
- add a diagnostic write, or
- use a global variable?

Use IO in a declarative language?

Design a declarative language?

IO changes state, implicitly or explicitly

In imperative languages IO changes state implicitly

Declarative languages expose the semantics, so state changes are naturally explicit

However, making everything explicit results in a lot of clutter

It makes code hard to write, modify and read

So syntactic sugar and other mechanisms are introduced to make state changes implicit again (and not just for IO)

Plus various methods to distinguish data structures which are single-threaded through the computation: monads (Haskell), unique types (Clean), unique modes (Mercury), sharing analysis (Mars), ...

Haskell uses monads and "do" notation

```
echo_char = do c <- getChar; putChar c
```

```
echo_char = getChar >>= putChar
```

The type of the instance of >>= reveals some of what has been made implicit

```
(>>=) :: IO Char -> (Char -> IO ()) -> IO ()
```

Passing multiple forms of state and converting from non-monadic code to monadic code are painful

Mercury uses DCG and state variable notation

```
echo_char(!IO) :- get_char(C, !IO), put_char(C, !IO).
```

```
echo_char --> get_char(C), put_char(C).
```

Both are syntactic sugar for

```
echo_char(IO_0, IO_2) :-
    get_char(C, IO_0, IO_1), put_char(C, IO_1, IO_2).
```

State variable notation is convenient for multiple forms of state but still somewhat verbose and argument order can be an issue

The pairs of arguments and their types and modes are generally made explicit in declarations

Pawns uses state variables (not Mercury)

State variables are declared independently of functions

They have a name and a type, which must be a (mutable) ref

```
!io :: ref(iotype).  % defined internally in Pawns
```

Functions are declared to use them as "implicit" arguments and/or results

In the definition of such functions they can be used and updated

Calls to such functions must be prefixed with "!"

```
echo_char(v) = {
    c = !get_char(void);
    !put_char(c)
}.

echo_char(v) = { !putchar(!getchar(void)) }.
```

Operations on state

State can be

- Created and written/initialized with a value
- Read, returning the current value
- Written/updated with a new value

Pawns function declarations say which operations the function performs

- `wo` (write only): the variable *must* be given a value
- `ro` (read only): the variable may be read (used) but not written/updated
- `rw` (read write): the variable may be read and also written

Semantically, the value is returned from the function, passed to the function, or both

Examples

```
get_char :: void -> int
    implicit rw io.
put_char :: int -> void
    implicit rw io.

!counter :: ref(int).

init_counter :: int -> void
    implicit wo counter.
init_counter(n) = {*counter = n}.

add_to_counter :: int -> void
    implicit rw counter.
add_to_counter(n) = {*!counter := *counter + n}.
```

Examples (cont.)

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

% Adds all nodes in tree to counter
count_tree :: tree -> void
    implicit rw counter.
count_tree(t) = {
    cases t of {
    case empty:
        return
    case node(l, n, r):
        !count_tree(l);
        !add_to_counter(n);
%       !count_tree(r)                    % version 1
        !add_to_counter(sum_tree(r))      % version 2
}   }.
```

Examples (cont.)

```
% Returns sum of nodes in tree
% Type signature guarantees it is purely declarative
sum_tree :: tree -> int.
sum_tree(t) = {
                          % counter undefined here
    !init_counter(0);     % binds counter
    !count_tree(t);       % uses/updates counter
    *counter              % returns final counter value
}.
```

Use of state is encapsulated

The recursive call to `sum_tree` from `count_tree` does not affect the state of `count_tree`

Implementation

Pawns state variables are not syntactic sugar for extra arguments

A state variable is a (tamed) global/static variable

Functions where it is declared implicit use it directly (with restrictions on updating); the explicit `ref` is optimized away

Functions where it is not declared implicit but which call a `wo` function save and restore its value using a local variable

Thus `count_tree` uses a static variable for the counter rather than an extra argument and return value. It is saved and restored using the stack frame for `sum_tree`

Implementation (cont.)

```
intptr_t  counter_VALUE;
#define counter (&counter_VALUE)
void  init_counter(intptr_t n);
void  add_to_counter(intptr_t n);
void  count_tree(tree t);
intptr_t
sum_tree(tree t) {
    intptr_t  _OLDVAL_counter= *counter;
    intptr_t V1 = 0;
    init_counter(V1);
    count_tree(t);
    intptr_t V3 = *counter;
    *counter = _OLDVAL_counter;
    return(V3);
}
```

Pros and cons

- \+ 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
- +/− Simple implementation uses fewer registers
- − Need for `wo` functions; could support eg, `!counter = 0`
- − Semantics not quite so clear
- − Code less flexible (based on state variable names, not just their types), including higher order
- − Type checking for higher order a bit more complicated

Summary

- Relatively minor changes to global variables “tame” them
- Encapsulate use of state inside “pure” code
- Creation/initialization of state important
- Stack/auto variables can be used to save/restore state instead of storing current value

Discussion

Questions/Comments . . . ?

Higher order/type checking

```
map_counter::(A->B implicit rw counter) -> list(A) -> list(B)
    implicit rw counter.
map_counter(f, xs) = {
    cases xs of {
    case nil:
        nil
    case cons(x, xs1):
        cons(!f(x), !map_counter(f, xs1))
    }
}.

mc1 :: list(int) -> list(void)
    implicit rw counter.
mc1(xs) = !map_counter(!add_to_counter, xs).
% mc1(xs) = !map_counter(!get_counter, xs).
```

Semantics (ignoring update details)

```
!s :: s_t.
frw :: a_t -> r_t implicit rw s.
fro :: a_t -> r_t implicit ro s.
...
    x = frw(a);
    y = fro(a);
% ==================>
frw :: pair(a_t,s_t) -> pair(r_t,s_t). % arg is mutable!
fro :: pair(a_t,s_t) -> r_t.
...
    v123 = frw(pair(a,s_42));  % s_42 is value of s here
    x = fst(v123);
    s_43 = snd(v123);          % s_43 is value of s here
    y = fro(pair(a,s_43));
```

Single threading (or not)

Global variables are naturally single-threaded

In Pawns we can have aliases for state variables but the sharing analysis of Pawns means any use/update of state variables is explicit

```
counter_alias :: void -> ref(int) implicit ro counter
    sharing counter_alias(v) = r
    pre nosharing
    post r = counter.       % must have alias in postcondition
counter_alias(v) = counter.

smash_counter :: int -> void
    implicit rw counter.    % must be rw
smash_counter(n) = {
    ca = !counter_alias(void);
    *!ca := n !counter}.    % update of counter made explicit
```