Playing with bird guts

Jonathan Worthington

YAPC::EU::2007

I'm not going to do a dissection.

The Plan For Today

- Take a simple Parrot program, written in PIR
- Look, from start to finish, at what happens when we feed it to the Parrot VM
- Then we'll cover a couple of little odds and ends that didn't really fit into that
- Do ask questions along the way if something isn't clear
- Don't throw up in here if you find the guts too disgusting, kplzthnx

The Example Program

```
.sub main :main
    $I0 = 1
loop:
    if $I0 > 10 goto exit
    $P0 = square_as_pmc($I0)
    say $P0
    inc $I0
    goto loop
exit:
.end
```

```
.sub square_as_pmc
    .param int x
    x = mul x, x
    $P0 = new 'Integer'
    $P0 = x
    .return($P0)
.end
```

IMCC

Intermediate Code Compiler

- We invoke Parrot to run this program:

```
./parrot example.pir
```

- And it enters IMCC, which is the default compiler front-end to Parrot
- Parrot does not interpret PIR directly, but instead compiles it to bytecode (like machine code, but for a virtual machine), which can be interpreted efficiently or compiled
- IMCC is the thing that does the PIR => bytecode translation

IMCC - Tokenization

- Breaks the PIR up into tokens

```
.sub main :main
    $I0 = 1
loop:
```

SUB | IDENT(main) | MAIN
IREG(0) | = | DIGIT(1)
LABEL(loop)

- Implemented using lex, a popular tokenizer generator; syntax along the lines of:

```
DIGIT                        [0-9]
%%
".sub"                       return(SUB);
<emit,INITIAL>":main"        return(MAIN);
<emit,INITIAL>\$I[0-9]+ DUP_AND_RET(valp, IREG);
```

IMCC - Parsing

- The parser takes the stream of tokens, attempts to match patterns of tokens and builds a data structure describing the program
- A program is described as a list of compilation units (one PIR sub results in one compilation unit)
- A unit in turn contains, amongst other things, a list of instructions

IMCC - Parsing

- The parser is written using yacc

```
sub: SUB {
        IMCC_INFO(interp)->cur_unit =
            imc_open_unit(interp, IMC_PCCSUB);
     }
     sub_label_op_c {
         iSUBROUTINE(interp,
             IMCC_INFO(interp)->cur_unit, $3);
     }
     sub_proto '\n' { ... }
     sub_params
     sub_body
     ESUB { ... }
     ;
```

IMCC - Parsing

- The parser is written using yacc

```
sub: SUB {
        IMCC_INFO(interp)->cur_unit =
            imc_open_unit(interp, IMC_PCCSUB);
     }
     sub_label_op_c {
         iSUBROUTINE(interp,
             IMCC_INFO(interp)->cur_unit, $3);
     }
     sub_proto '\n' { ... }
     sub_params
     sub_body
     ESUB { ... }
     ;
```

This is the SUB token

IMCC - Parsing

- The parser is written using yacc

```
sub: SUB {
        IMCC_INFO(interp)->cur_unit =
            imc_open_unit(interp, IMC_PCCSUB);
     }
     sub_label_op_c {
         iSUBROUTINE(interp,
             IMCC_INFO(interp)->cur_unit, $3);
     }
     sub_proto '\n' { ... }
     sub_params
     sub_body
     ESUB { ... }
     ;
```

Chunk of C we run – adds a new compilation unit to the list

IMCC - Parsing

- The parser is written using yacc

```
sub: SUB {
         IMCC_INFO(interp)->cur_unit =
             imc_open_unit(interp, IMC_PCCSUB);
     }
     sub_label_op_c {
         iSUBROUTINE(interp,
             IMCC_INFO(interp)->cur_unit, $3);
     }
     sub_proto '\n' { ... }
     sub_params
     sub_body
     ESUB { ... }
     ;
```

Makes the unit a sub and associates the provided name with it

IMCC - Parsing

- The parser is written using yacc

```
sub: SUB {
        IMCC_INFO(interp)->cur_unit =
            imc_open_unit(interp, IMC_PCCSUB);
     }
     sub_label_op_c {
         iSUBROUTINE(interp,
             IMCC_INFO(interp)->cur_unit, $3);
     }
     sub_proto '\n' { ... }
     sub_params
     sub_body
     ESUB { ... }
     ;
```

These refer not to tokens, but other grammar rules

IMCC – Register Allocation

- When we write PIR, we can use virtual registers

```
$I0 = 42
say $I0
$I1 = 666
say $I1
```

→

```
I0 = 42
say I0
I0 = 666
say I0
```

- However, we only really need one integer register
- Register allocation algorithm tries to minimise the number of registers used (to always get the minimum is NP-complete)

IMCC – Optimization

- There's a whole load of optimizations that we can perform on register code
- Since CPUs are register architectures, there has been much research done on this
- The optimizer is not run by default at the moment, beyond some simple constant folding
- Implements various techniques, though still plenty of room for improvements

IMCC – Bytecode Generation

- We translate the in-memory data structure to bytecode – a stream of integers
- Each instruction has an instruction code, which we emit first
- This is followed by its operands
  - Register number, immediate integer constant or index into the constants table

```
$I0 = 1
if $I0 > 10 goto exit
```

→

```
796 0 1
215 10 0 31
```

IMCC – Bytecode Files (aka Packfiles)

- Can write the bytecode stream, along with the table of constants and debug information, out to disk
- Store it with the byte ordering and word size of the machine it was compiled on => no decoding needed when loading on same architecture => can memory map the file!
- However, if we load it on a different architecture, it has the information it needs to re-order bytes or change word size

Initialization

Memory Pools

- PMCs and STRINGs are garbage collectable
- Allocated out of fixed sized pools of objects
- Keep a list of pools for each size
- If a pool gets full and we need more objects of that size, create another one and add it to the list
- At startup, we allocate memory for these memory pools

Contexts

- We create one context per invocation of a sub or closure
  - A sub that calls itself recursively 10 times will result in 10 contexts
- A Context may reference other Contexts
  - Caller context – the dynamic chain
  - Outer context – the static chain (where one sub or closure is textually enclosed within another)

Contexts

- Context data structure contains, amongst other things…
  - Pointers into the register set

I Registers	N Registers	S Registers	P Registers

bp   bp_sp

  - The number of registers used by the current sub/closure
  - A pointer to the current Sub or Closure
  - A pointer to the current return continuation – more about this later

Contexts

- We create an initial, empty context that doesn't refer to any subroutine
  - Need this so main has a context to return into
- Then we call the invoke v-table method of the main subroutine
  - Creates a context for the current invocation of itself
  - Returns bytecode offset of main sub

Execution Time!

Enter The Runloop!

- A runloop executes the instruction at the current program counter, until the end of the program is reached (or uncaught exception)
- There's more than one runloop; the simplest one has:
  - One C function for each instruction
  - An array of pointers to these functions
- Index into the array with the instruction code to locate the function to call

Our First Instruction

- Our first instruction is an integer assignment

```
$I0 = 1
```

- This compiles down to the instruction set_i_ic (first operand is an integer register, the second is an integer constant)
- Function implementing the opcode generated from entry in a .ops file

```
inline op set(out INT, in INT) :base_core {
  $1 = $2;
  goto NEXT();
}
```

Our First Instruction

- Our first instruction is an integer assignment

```
$I0 = 1
```

- This compiles down to the instruction set_i_ic (first argument is an integer register, the second is an integer constant)
- Function implementing the opcode generated from entry in a .ops file

```
inline op set(out INT, in INT) :base_core {
  $1 = $2;
  goto NEXT();
}
```

Types of registers the op operates on

Our First Instruction

- Our first instruction is an integer assignment

```
$I0 = 1
```

- This compiles down to the instruction set_i_ic (first argument is an integer register, the second is an integer constant)
- Function implementing the opcode generated from entry in a .ops file

```
inline op set(out INT, in INT) :base_core {
  $1 = $2;
  goto NEXT();
}
```

C code; $n refers to the nth operand

Our First Instruction

- Our first instruction is an integer assignment

```
$I0 = 1
```

- This compiles down to the instruction set_i_ic (first argument is an integer register, the second is an integer constant)
- Function implementing the opcode generated from entry in a .ops file

```
inline op set(out INT, in INT) :base_core {
  $1 = $2;
  goto NEXT();
}
```

Gets substituted by ops build tool

Our Second Instruction

- A conditional branch

```
if $I0 > 10 goto exit
```

- This compiles down to the instruction lt_ic_i_ic (< is > but swap the operands)
- Label compiles down to offset (in words)

```
inline op lt(in INT, in INT, labelconst INT)
:base_core {
  if ($1 < $2) {
    goto OFFSET($3);
  }
  goto NEXT();
}
```

Calling Subroutines

Compiling Calls

- The call instruction:

```
$P0 = square_as_pmc($I0)
```

- Actually compiles down to several instructions when translating the PIR to bytecode:

```
set_args PC4, I0
set_p_pc P0, square_as_pmc
get_results PC7, P1
invokecc P0
```

Compiling Calls

- set_args specifies the registers containing the arguments to be passed

```
set_args PC4, I0
```

  - PC4 refers to a PMC in the constants table, which specifies the signature
  - The opcode takes a variable number of operands
- get_results works the same way – note that we do this before the call

```
get_results PC7, P1
```

Compiling Calls

- Looking up the sub to call and invoking it are two separate steps => allows sub refs to work
- Sub PMC representing the sub to call is in the constants table – look it up and store it in P0

```
set_p_pc P0, square_as_pmc
```

- Then, with everything set up, use the invokecc opcode to do the call

```
invokecc P0
```

Inside The Callee

- The parameter syntax:

```
.param int x
```

- Actually compiles down to the get_params opcode:

```
get_params PC4, I0
```

  - Once again, specifies a signature and registers to receive the arguments
  - When we execute this op, we actually do the passing – that is, copy the values from the caller's to the callee's registers

Inside The Callee

- Similarly, the return syntax:

```
.return(P0)
```

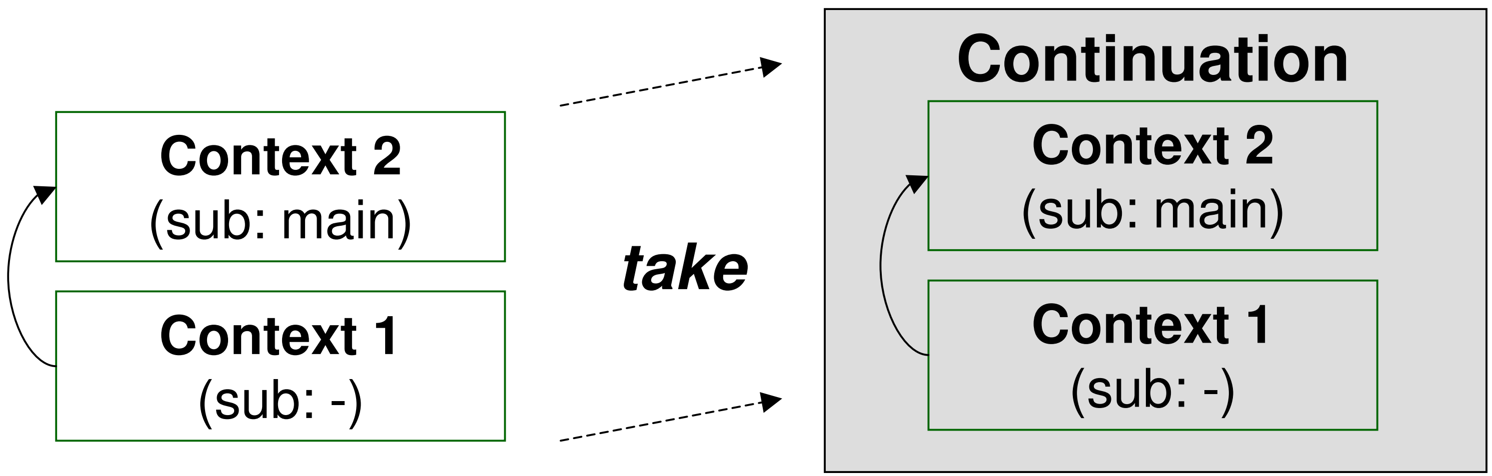
- Compiles down to the set_returns opcode:

```
set_returns PC7, P0
```

  - The caller already specified the registers to store the results in
  - When we execute this op, we do the returning – storing the values from the callee's registers into the caller's registers.

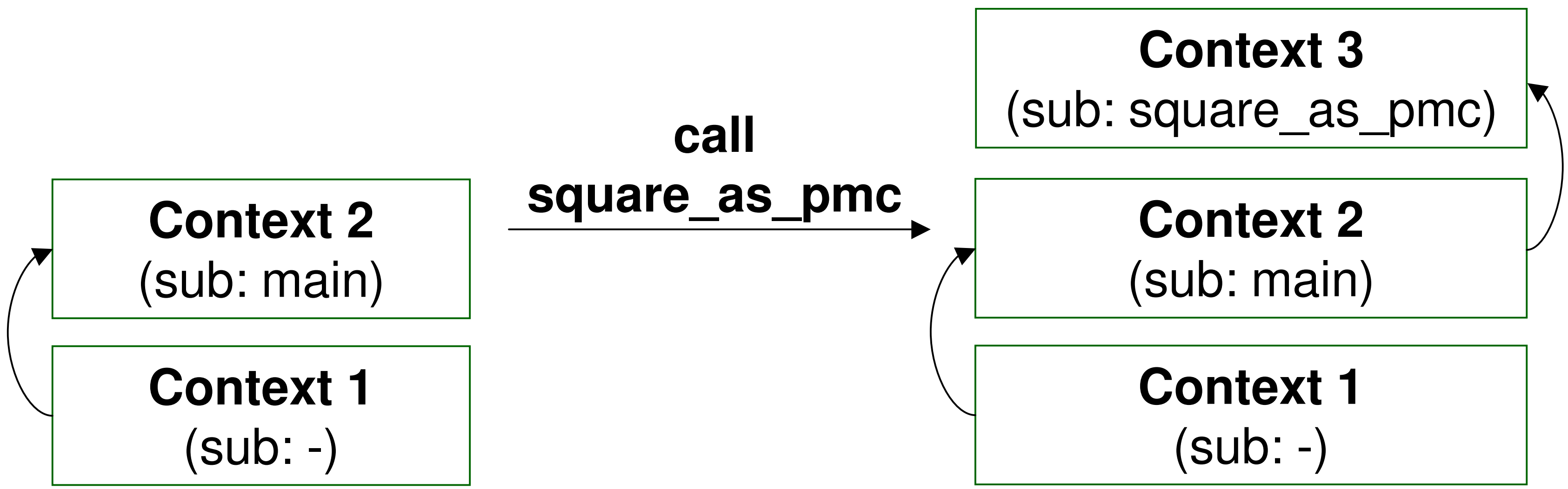
Continuation Passing Style

- When we take a continuation, we make a copy (lazily) of the current (dynamic) chain of contexts and the current program counter



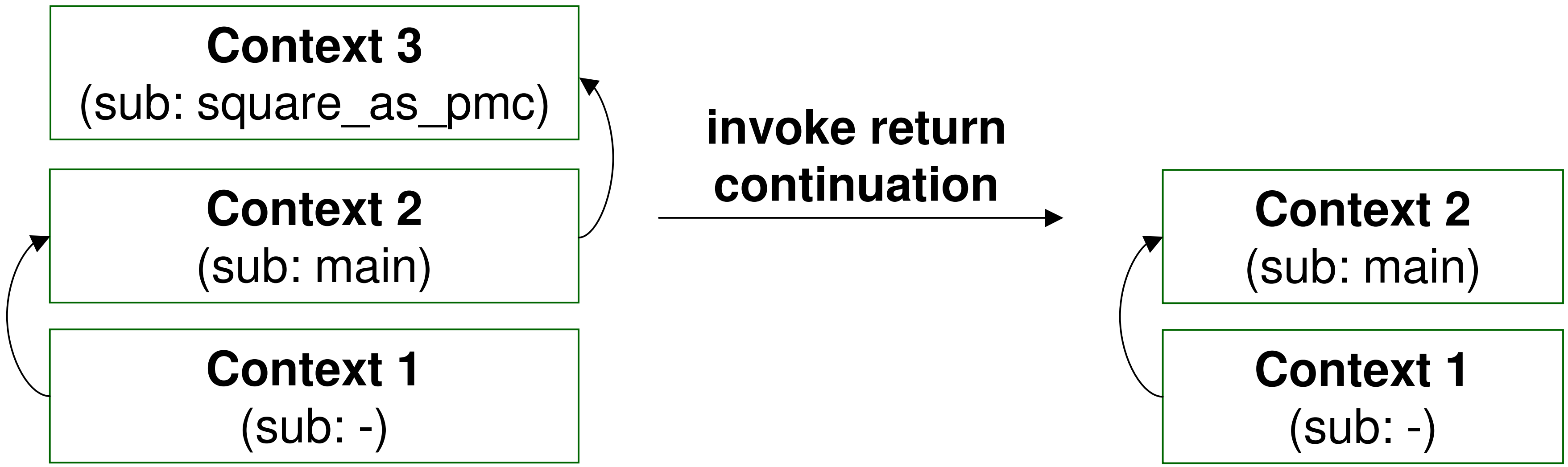
Continuation Passing Style

- When we are making a call, we first make a continuation (called a return continuation)
- Then we create the context for the sub being called and store the continuation inside it



Continuation Passing Style

- The .return(PO) in PIR actually compiles down to two instructions – a `set_returns` and a `returncc`
- `returncc` invokes the return continuation, which restores the call chain and PC it took



PMCs

Looking At `square_as_pmc`

- The square_as_pmc code is the first time we explicitly have dealt with PMCs (although some of the generated code we saw earlier dealt with them too)

```
.sub square_as_pmc
    .param int x
    x = mul x, x
    $P0 = new 'Integer'
    $P0 = x
    .return($P0)
.end
```

Looking At `square_as_pmc`

- The square_as_pmc code is the first time we explicitly have dealt with PMCs (although some of the generated code we saw earlier dealt with them too)

```
.sub square_as_pmc
    .param int x
    x = mul x, x
    $P0 = new 'Integer'
    $P0 = x
    .return($P0)
.end
```

Instantiate a PMC

Looking At `square_as_pmc`

- The square_as_pmc code is the first time we explicitly have dealt with PMCs (although some of the generated code we saw earlier dealt with them too)

```
.sub square_as_pmc
    .param int x
    x = mul x, x
    $P0 = new 'Integer'
    $P0 = x
    .return($P0)
.end
```

Assign to a PMC

Playing with bird guts

PMCs

- Classes implemented in C
- Written in a .pmc file, which is run through a preprocessor to generate .c and .h files
- Implement some subset of a set of vtable methods

```
pmclass Integer extends scalar {
    void init() {
        PMC_int_val(SELF) = 0;

    }
    ...
}
```

PMCs

- Classes implemented in C
- Written in a .pmc file, which is run through a preprocessor to generate .c and .h files
- Implement some subset of a set of vtable methods

```
pmclass Integer extends scalar {
    void init() {
        PMC_int_val(SELF) = 0;
    }
    ...
}
```

Name of the class

Playing with bird guts

PMCs

- Classes implemented in C
- Written in a .pmc file, which is run through a preprocessor to generate .c and .h files
- Implement some subset of a set of vtable methods

```
pmclass Integer extends scalar {
    void init() {
        PMC_int_val(SELF) = 0;
    }
    ...
}
```

Inheritance

PMCs

- Classes implemented in C
- Written in a .pmc file, which is run through a preprocessor to generate .c and .h files
- Implement some subset of a set of vtable methods

```
pmclass Integer extends scalar {
    void init() {
        PMC_int_val(SELF) = 0;
    }
    ...
}
```

Implementation of init vtable method

PMCs

- Classes implemented in C
- Written in a .pmc file, which is run through a preprocessor to generate .c and .h files
- Implement some subset of a set of vtable methods

```
pmclass Integer extends scalar {
    void init() {
        PMC_int_val(SELF) = 0;
    }
    ...
}
```

The invocant

PMCs

- Classes implemented in C
- Written in a .pmc file, which is run through a preprocessor to generate .c and .h files
- Implement some subset of a fixed set of vtable methods

```
pmclass Integer extends scalar {
    void init() {
        PMC_int_val(SELF) = 0;
    }
    ...
}
```

Macro accessing a slot in the PMC

PMC Instantiation

- The new opcode instantiates a PMC

```
$P0 = new 'Integer'
```

  - Looks up the name and resolves it to a PMC type number (in the future, may do lookup via the namespace)
  - Provided it's found, get a chunk of memory from a memory pool
  - Initialize the PMC data structure
  - Call the `init` vtable method

Calling vtable methods

- Opcode implementations for PMCs simply call the vtable methods

```
$P0 = x # x is an integer parameter
```

- In this case, it's the set_p_i opcode

```
inline op set(invar PMC, in INT) :base_core {
  $1->vtable->set_integer_native(interp, $1, $2);
  goto NEXT();
}
```

- Note that we pass the interpreter and the PMC that will be accessible in the method through SELF – implicit when writing PMC

Calling vtable methods

- And here's the vtable method in integer.pmc that we end up calling:

```
void set_integer_native(INTVAL value) {
    PMC_int_val(SELF) = value;
}
```

- It's not quite this simple for all vtable methods – some do multiple dispatch
  - For example, when implementing the `add` vtable method, the other PMC we are to add may not be an `Integer` PMC – need to handle this correctly

Garbage Collection

When The Memory Pools Are Full…

- One of the steps for instantiating a PMC is getting a chunk of memory from one of the memory pools
- If all the pools are full, we do a garbage collection run
  - Find PMCs that are no longer in use and add them to the free list
- If that fails to provide us with more memory, we allocate another pool

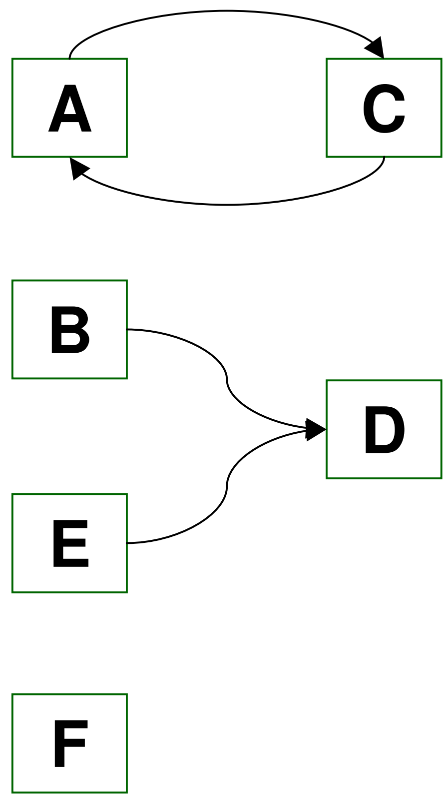
For Our Program...

- Looking at our main routine, we see that a PMC only lasts a single iteration of the loop

```
.sub main :main
    $I0 = 1
loop:
    if $I0 > 10 goto exit
    $P0 = square_as_pmc($I0)
    say $P0
    inc $I0
    goto loop
exit:
.end
```

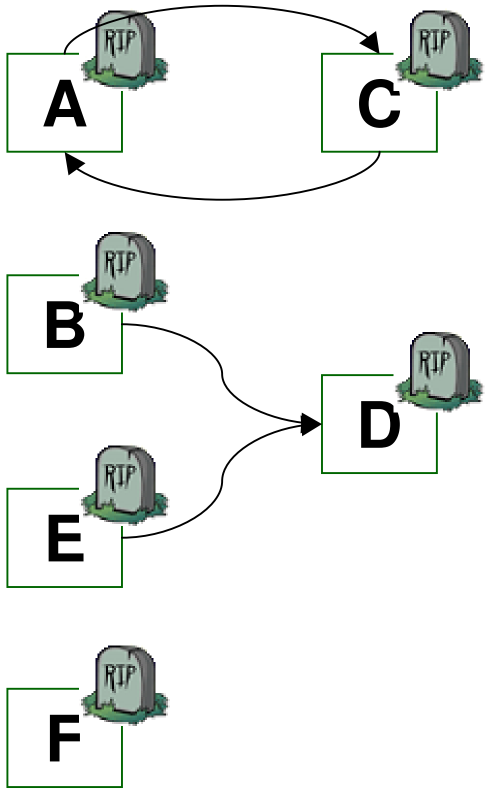
A More Interesting Example

- Here, arrows represent PMCs referencing each other



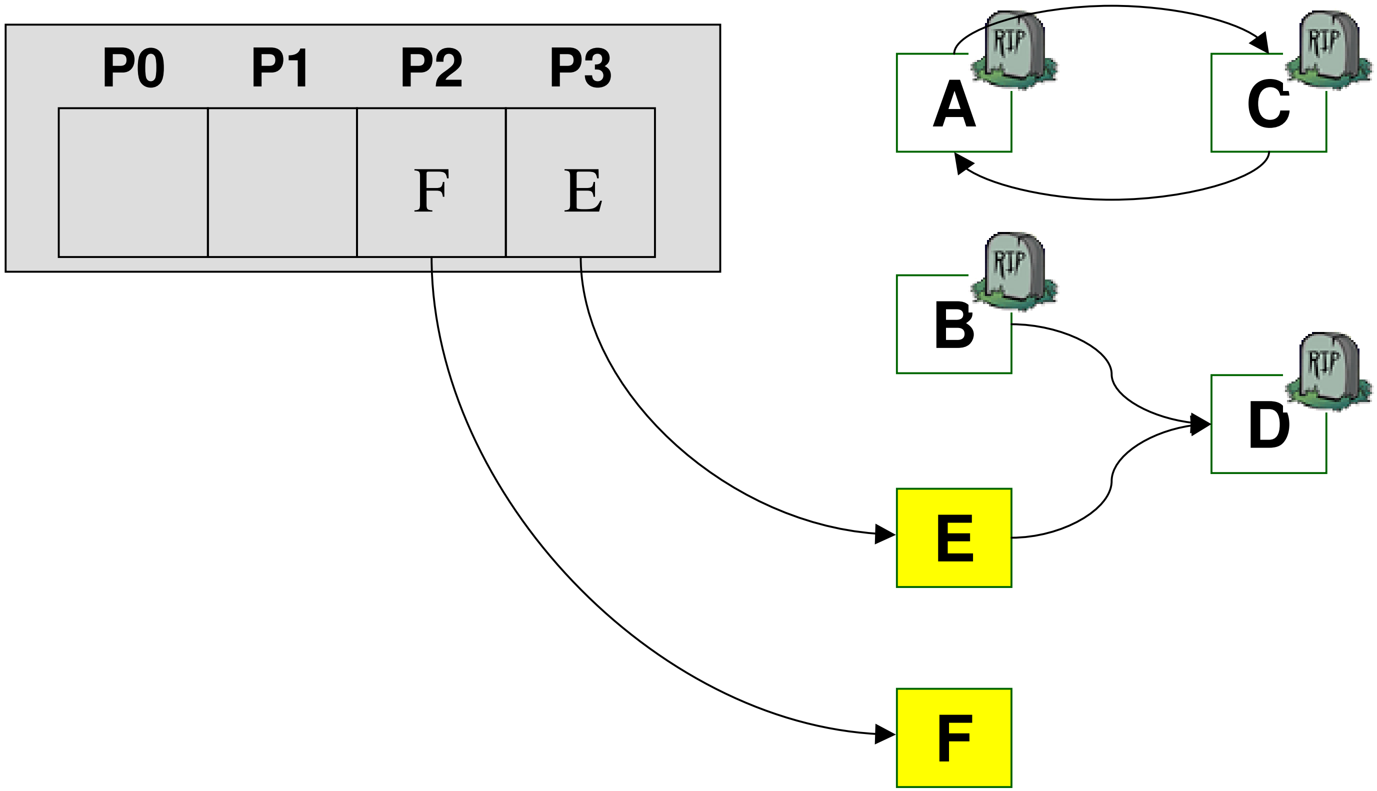
Dead Object Detection

- At the start of DOD, we assume that all objects are unreachable or "dead"



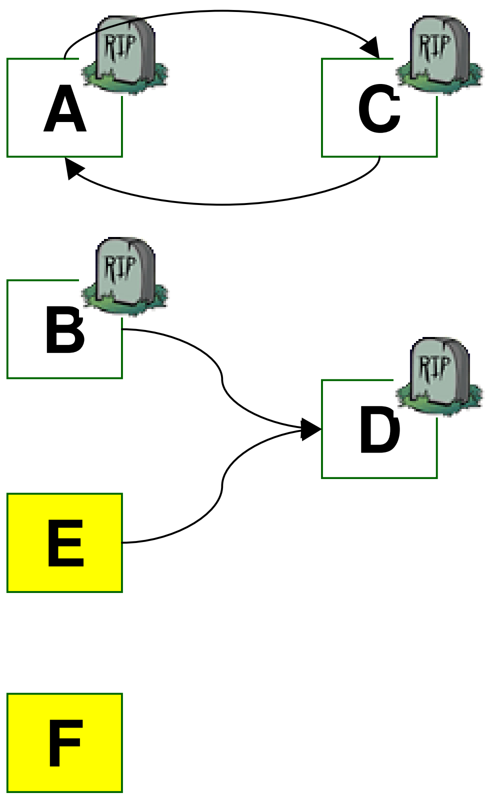
Dead Object Detection

- Then look through the registers to see what PMCs are referenced from there



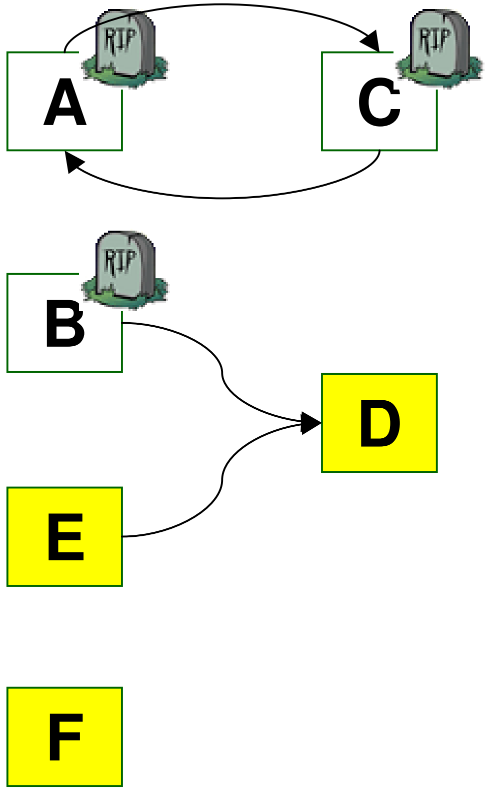
Dead Object Detection

- Then we iteratively locate all PMCs referenced by living PMCs



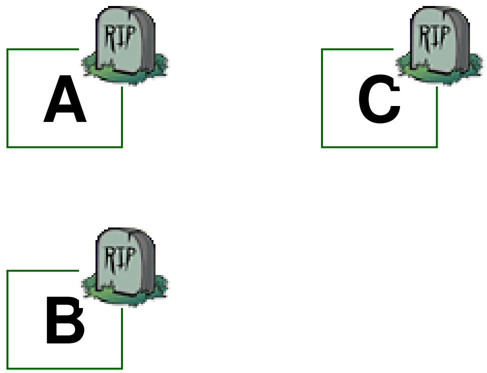
Dead Object Detection

- Then we iteratively locate all PMCs referenced by living PMCs



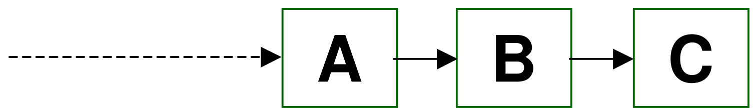
Dead Object Detection

- All objects that are have not been marked alive by this point are unreachable



Sweep

- The objects that were found to be dead can now be put on the free list



- Then they are available to allocate more PMCs with
- This is a simple mark-and-sweep scheme – there are more complex approaches that have been prototyped in Parrot.

JIT

What is a JIT compiler?

- Just In Time means that a chunk of bytecode is compiled when it is needed.
- Compilation involves translating Parrot bytecode into machine code understood by the hardware CPU.
- High performance – can execute some Parrot instructions with one CPU instruction.
- Not at all portable – custom implementation needed for each type of CPU.

How does JIT work?

- For each CPU, write a set of macros that describe how to generate native code for the VM instructions.
  - Do not need to write these for every instruction; can fall back on calling the C function that implements it.
- A Configure script determines the CPU type and selects the appropriate JIT compiler to build if one is available.

How does JIT work?

- A chunk of memory is allocated and marked executable if the OS requires this.
- For each instruction in the chunk of bytecode that is to be translated:
    - If a JIT macro was written for the instruction, use that to emit native code.
    - Otherwise, insert native code to call the C function implementing that method, as an interpreter would.

The End

Thank You

Questions?