Parrot
What, where and why?

Jonathan Worthington

London Perl Workshop 2005

Parrot – What, where and why?

<u>A Multi-threaded Talk</u>

Asking and answering three questions – in parallel!

What?

What is Parrot? What does it do?

Where?

Where are we at with developing Parrot?

Why?

Why is Parrot designed the way it is?

Parrot – What, where and why?

What is Parrot?

- A runtime for dynamic languages.
  - Spawned by the need for a runtime engine for Perl 6.
  - Aims to provide support for many languages and allow interoperability between them.
- A register based virtual machine.
- Named after an April Fool’s joke.

Parrot – What, where and why?

Where are we with Parrot?

•Public development started in September 2001.

•Many of Parrot’s core features are now working, though several important subsystems not completely implemented or in some cases not specified.

•Pugs (the Perl 6 prototype interpreter) can target Parrot for some language features, and a number of other compilers underway.

Parrot – What, where and why?

We have the JVM & .NET CLR - why Parrot?

- .NET and the JVM built with static languages in mind; Perl, Python, etc. are dynamic and less well supported.
- .NET constrains high level semantics of languages to achieve interoperability. Parrot has interoperability provided at an assembly level – more later.
- Need to support the range of platforms that Perl 5 did, and more.

Parrot – What, where and why?

Parrot is a Virtual Machine

•Hides away the details of the underlying hardware platform and operating system.

•Defines a common set of instructions and a common API for I/O, threading, etc.

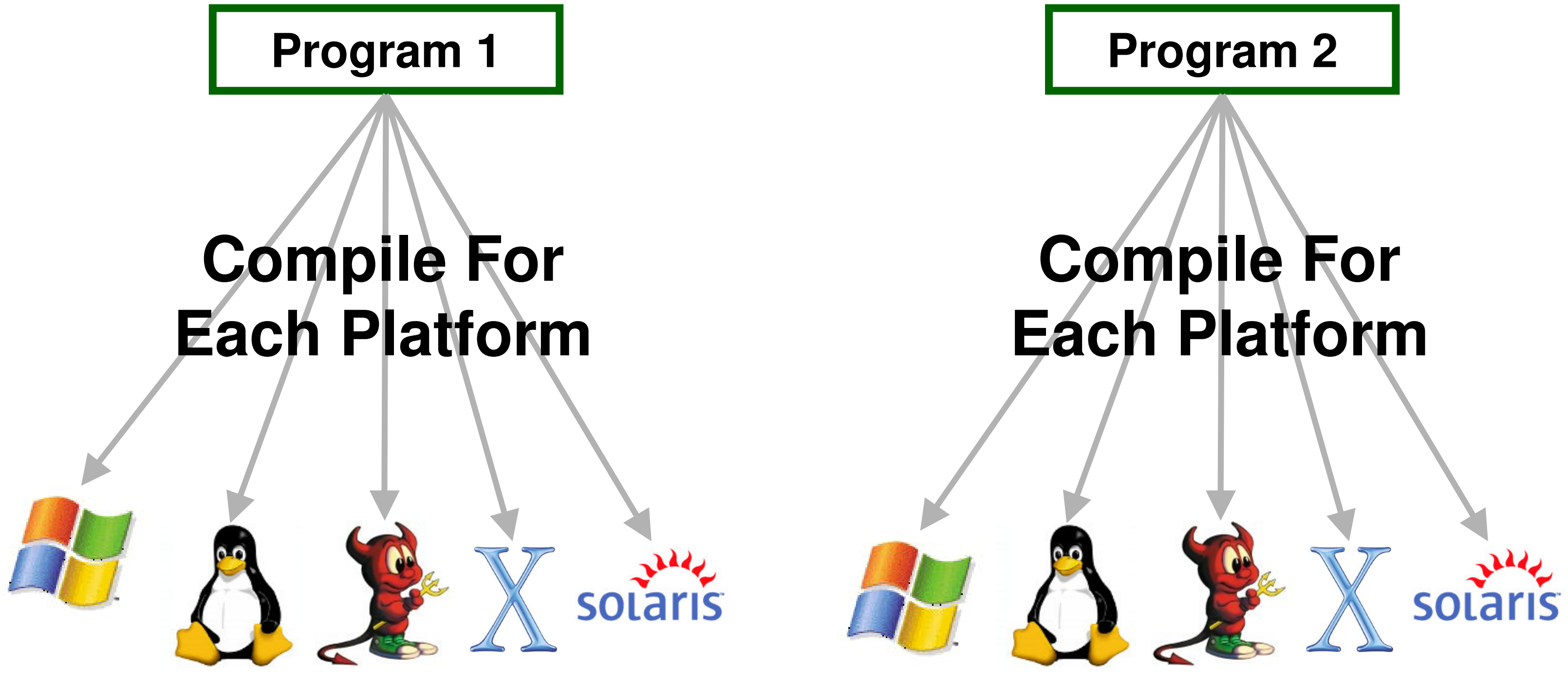
•Efficiently translates the virtual instructions to those supported by the underlying hardware and maps the common API to the one provided by the operating system.

•Supports high level language constructs.

Parrot – What, where and why?

Why Virtual Machines?

1. Simplified software development and deployment.

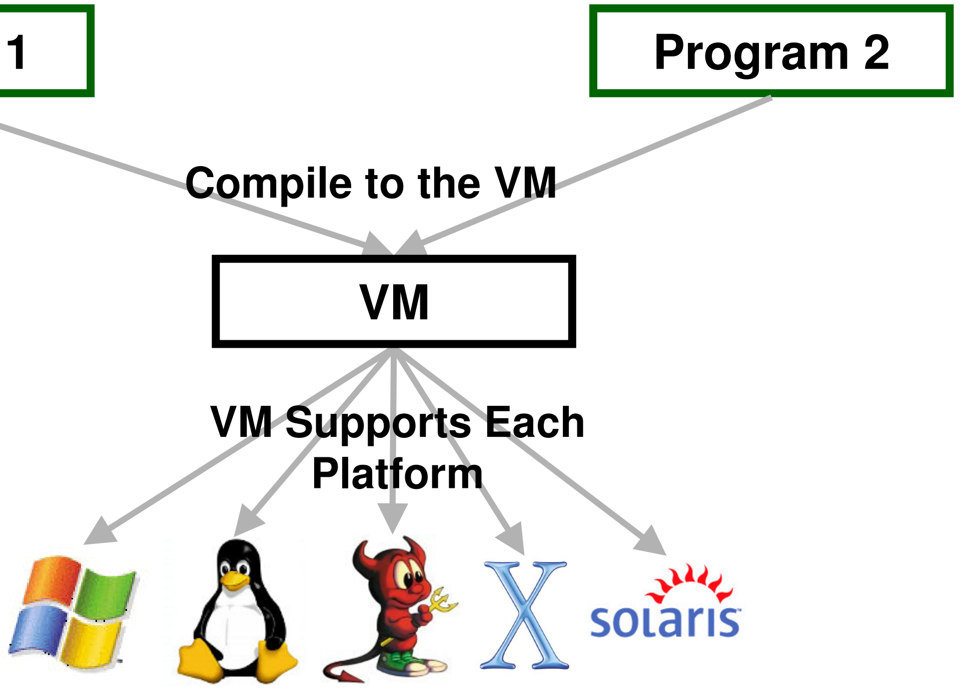


Without a VM

Parrot – What, where and why?

Why Virtual Machines?

1. Simplified software development and deployment.



With a VM

Parrot – What, where and why?

Why Virtual Machines?

2. High level languages have a lot in common.

- Strings, arrays, hashes, references, …
- Subroutines, objects, namespaces, …
- Closures and continuations
- Memory management

Can implement these just once in the VM.

Parrot – What, where and why?

Why Virtual Machines?

3. High level language interoperability becomes easier.

- A consistent way to call subroutines and methods.
- A common representation of data types: strings, arrays, objects, etc.
- Code in multiple languages essentially runs as a single program.

Parrot – What, where and why?

Why Virtual Machines?

4. Can provide fine grained security and quota restrictions.
    - "This program can connect to server X, but can not access any local files."
5. Debugging and profiling more easily supported.
6. Possibility of dynamic optimizations by exploiting what can be known at runtime but not at compile time.

Parrot – What, where and why?

Parrot is a Register Machine

- A register is a numbered location where working data can be stored.
- Most Parrot instructions either
  - Load data into registers from elsewhere
  - Perform operations on data held in registers (add, mul, and, or, …)
  - Compare values in registers (ifgt, ifle, …)
  - Store data from registers to elsewhere

Parrot – What, where and why?

Parrot is a Register Machine

The add instruction in Parrot adds the values stored in two registers and stores the result in a third.

add I1, I3, I4

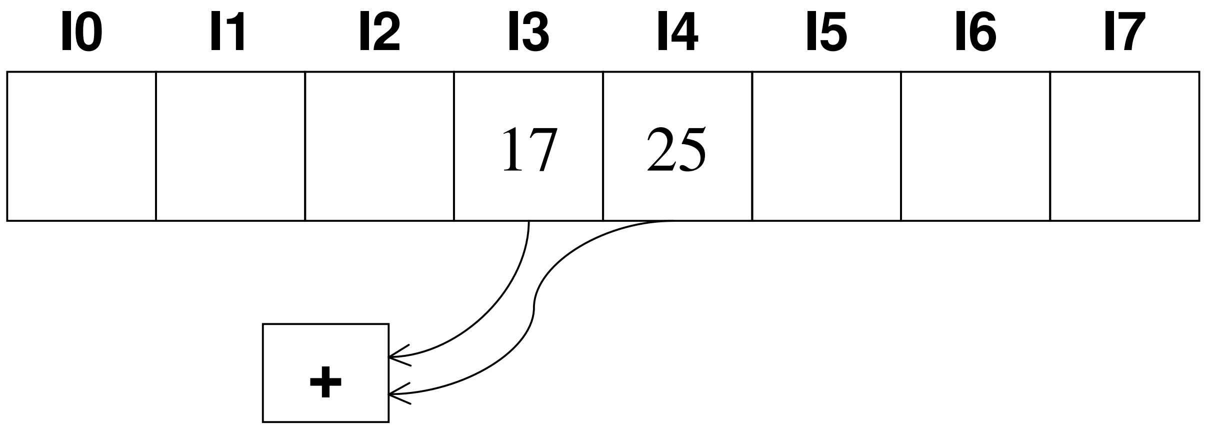
I0	I1	I2	I3	I4	I5	I6	I7
			17	25			

Parrot – What, where and why?

Parrot is a Register Machine

The add instruction in Parrot adds the values stored in two registers and stores the result in a third.

add I1, I3, I4

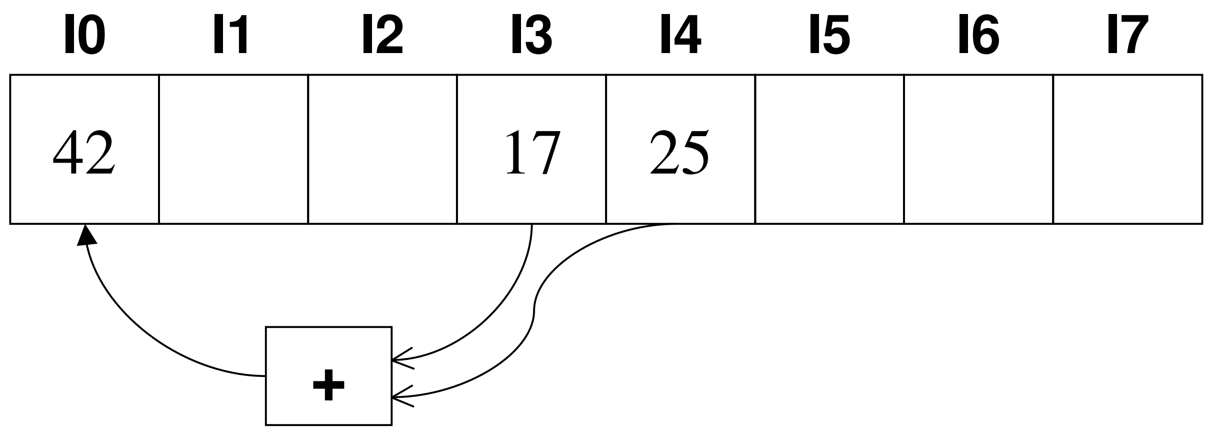


Parrot – What, where and why?

Parrot is a Register Machine

The add instruction in Parrot adds the values stored in two registers and stores the result in a third.

add I0, I3, I4



Parrot – What, where and why?

Why a register machine?

Many virtual machines, including .NET and JVM, are implemented as stack machines.

```
push 17

push 25

add
```

Parrot – What, where and why?

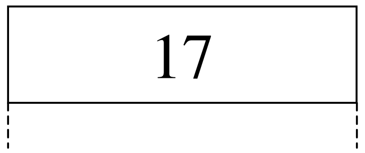
Why a register machine?

Many virtual machines, including .NET and JVM, are implemented as stack machines.

push 17

push 25

add



Parrot – What, where and why?

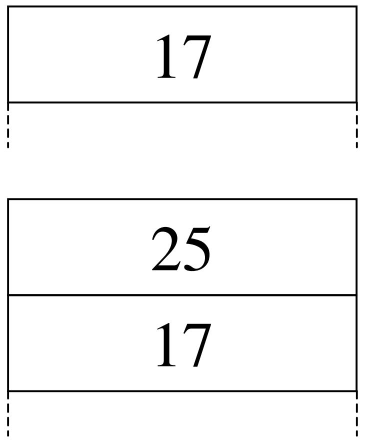
Why a register machine?

Many virtual machines, including .NET and JVM, are implemented as stack machines.

push 17

push 25

add



Parrot – What, where and why?

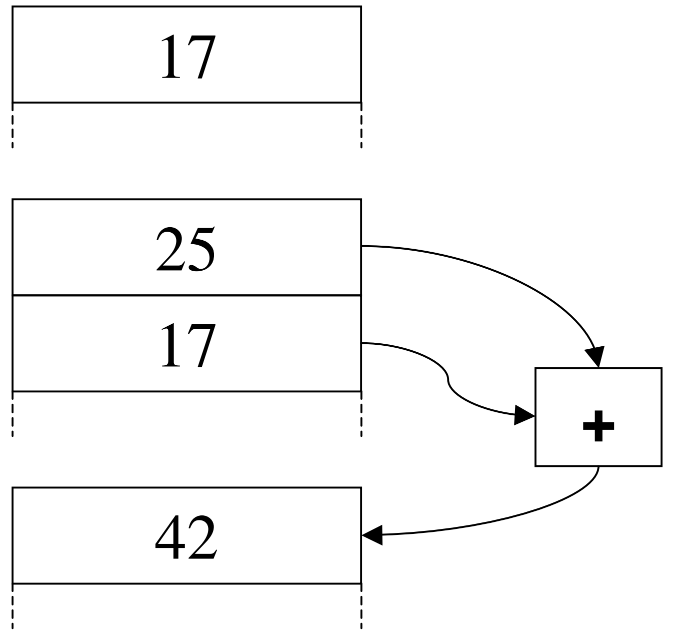
Why a register machine?

Many virtual machines, including .NET and JVM, are implemented as stack machines.

push 17

push 25

add



Parrot – What, where and why?

Why a register machine?

•What could be expressed in one register instruction took at least three stack instructions.

•When interpreting code, there is overhead for mapping each virtual instructions to a real one, so less instructions is a Good Thing.

•Also, no need for the interpreter to maintain a stack pointer.

Parrot – What, where and why?

Register Types

- Parrot has 4 types of register.
  - **I**nteger registers store native integers
  - **N**umber registers store native floating point numbers (probably doubles)
  - **S**tring registers store references to strings
  - **P**MC registers store references to Parrot Magic Cookies (more later)

Parrot – What, where and why?

Why Have Different Register Types?

- Need to provide the possibility of high performance execution
  - Native integer and floating point registers map directly to hardware.
- Also need to provide support for language specific behaviour and consistent cross-platform behaviour.
  - PMCs allow for implementation of types with custom behaviours.

Parrot – What, where and why?

Variable Sized Register Frames

- Registers in hardware CPUs are physical chunks of memory on the CPU, and there are a fixed number of them.
- Initially Parrot followed this, having 32 of each type of register making up a register frame.
- If more registers were needed an array stored in a PMC register could be used to spill values to.

Parrot – What, where and why?

Variable Sized Register Frames

- Parrot register frames are simply arrays located in main system memory.
- Therefore the restrictions on a hardware CPU need not apply to Parrot.
- Parrot has had variable sized register frames since release 0.3.1 (November ’05).
  - The number of registers of each type is simply what is used by a unit of code (a unit usually being a subroutine).

Parrot – What, where and why?

Why Variable Sized Register Frames?

- Never run out of registers so no need to spill, leading to faster execution.
- Units that only use a few registers will use less memory – especially good for deeply recursive code.
- The change could be done without breaking most existing Parrot programs.
- Downside is that the variable size of register frames adds a little “bookkeeping” overhead.

Parrot – What, where and why?

What do Parrot programs look like?

Parrot programs are mostly represented in one of three forms.

Best For People

PIR = Parrot Intermediate Representation

PASM = Parrot Assembly

PBC = Parrot Bytecode

Best For The VM

Parrot – What, where and why?

What does PIR look like?

```
.sub factorial
    .param int n
    .local int result

    if n > 1 goto recurse
    result = 1
    goto return

recurse:
    $I0 = n - 1
    result = factorial($I0)
    result *= n

return:
    .return (result)
.end
```

Simple sub declaration

Simple param. access syntax

Named registers

Register code looks like HLL

Virtual registers

Simple sub calling syntax

Simple return syntax

Parrot – What, where and why?

What does PASM look like?

```
factorial:
    get_params "(0)", I1
    lt 1, I1, recurse
    set I0, 1
    branch return
recurse:
    sub I2, I1, 1
@pcc_sub_call_0:
    set_args “(0)”, I2
    set_p_pc P0, factorial
    get_results “(0)”, I1
    invokecc P0
    mul I0, I1
return:
@pcc_sub_ret_1:
    set_returns “(0)”, I0
    returncc
```

Opcode to get parameters

Looks like assembly

Calling conventions exposed

Opcodes for returning

Parrot – What, where and why?

What does PBC look like?

- A portable binary file format.
    - Written with the endianness and word size of the machine that generated it – good for performance.
    - If running on a different type of machine translation done “on the fly” – good for portability.
- Can be executed (almost) directly by the Parrot virtual machine.

Parrot – What, where and why?

Why PIR, PASM and PBC?

•Need something that is efficient to load and directly execute – **PBC**

•Need something small to distribute – **PBC**

•Need something that is human readable and writable. – **PIR or PASM**

•Need a way to abstract away details (like calling conventions) from compilers – **PIR**

•Need low level assembly language – **PASM**

Parrot – What, where and why?

Where are we at with PIR/PASM/PBC?

•They all work and can be used.

•More PIR syntax still to come.

•PIR compiler needs some further tidying.

•Room for improvements to PIR optimization.

•PBC file format missing the ability to store some things, like HLL debug info and source.

•Need to provide support for working with PBC files from PIR.

Parrot – What, where and why?

What is a PMC?

- A PMC defines a type with a certain set of behaviours.
- Implements some of a pre-defined set of methods that represent behaviours a type may need to customize, such as integer assignment, addition or getting the number of elements.
- Method bodies written in C, but much code is generated by a PMC build too.

Parrot – What, where and why?

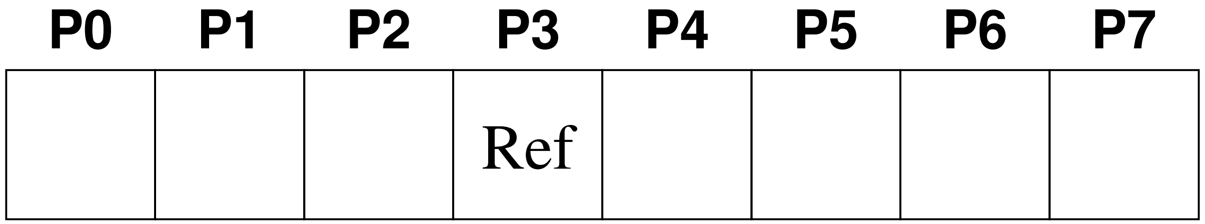
How do PMCs work?

- Each PMC has a pointer to a v-table.
  - A v-table is a list of function pointers to the code implementing each method of the PMC.
- When operations are performed on PMCs, the v-table is used to call the appropriate PMC method.
- Essentially, PMCs inherit from a base class and implement methods as needed.

Parrot – What, where and why?

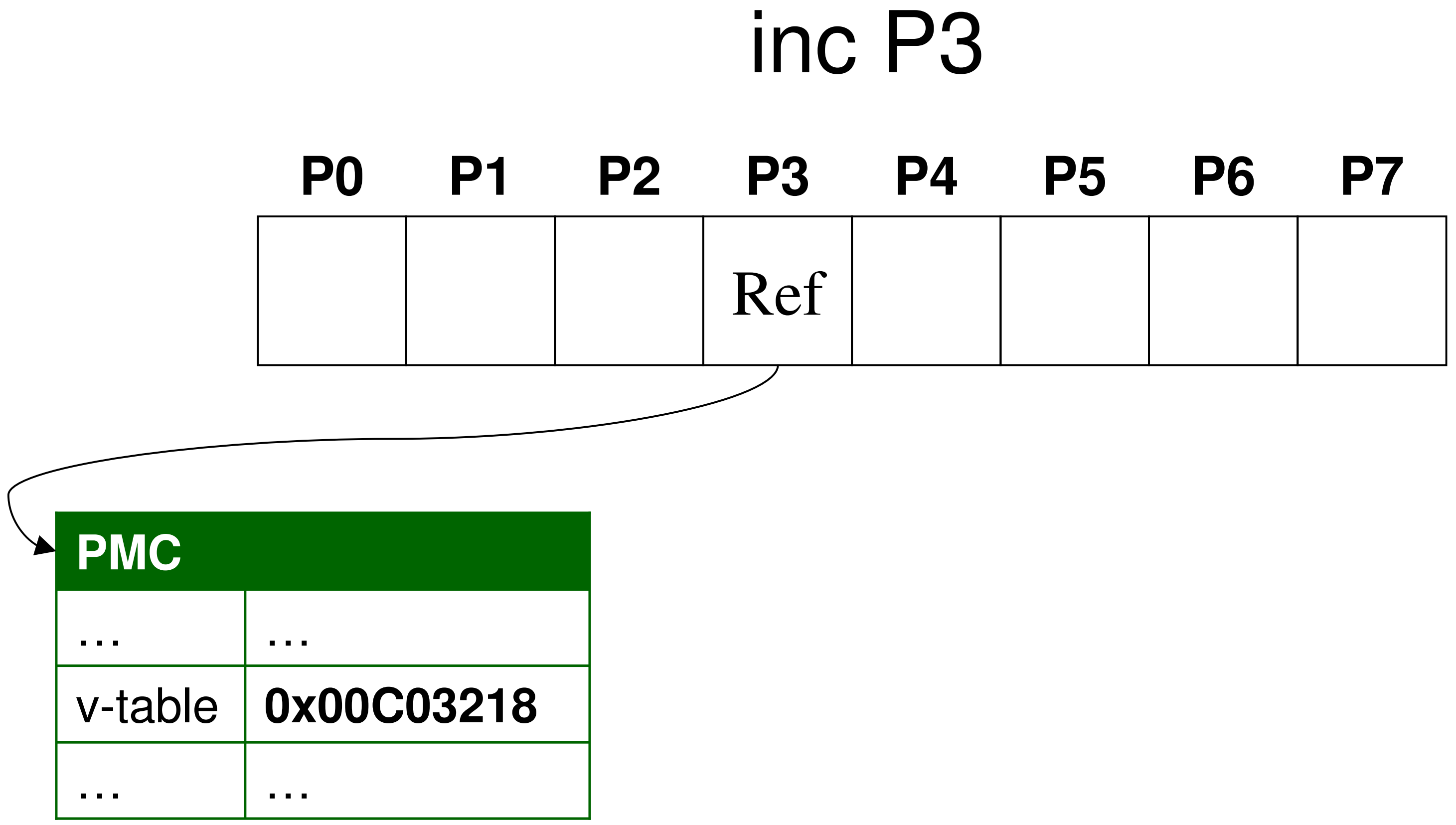
How do PMCs work?

inc P3



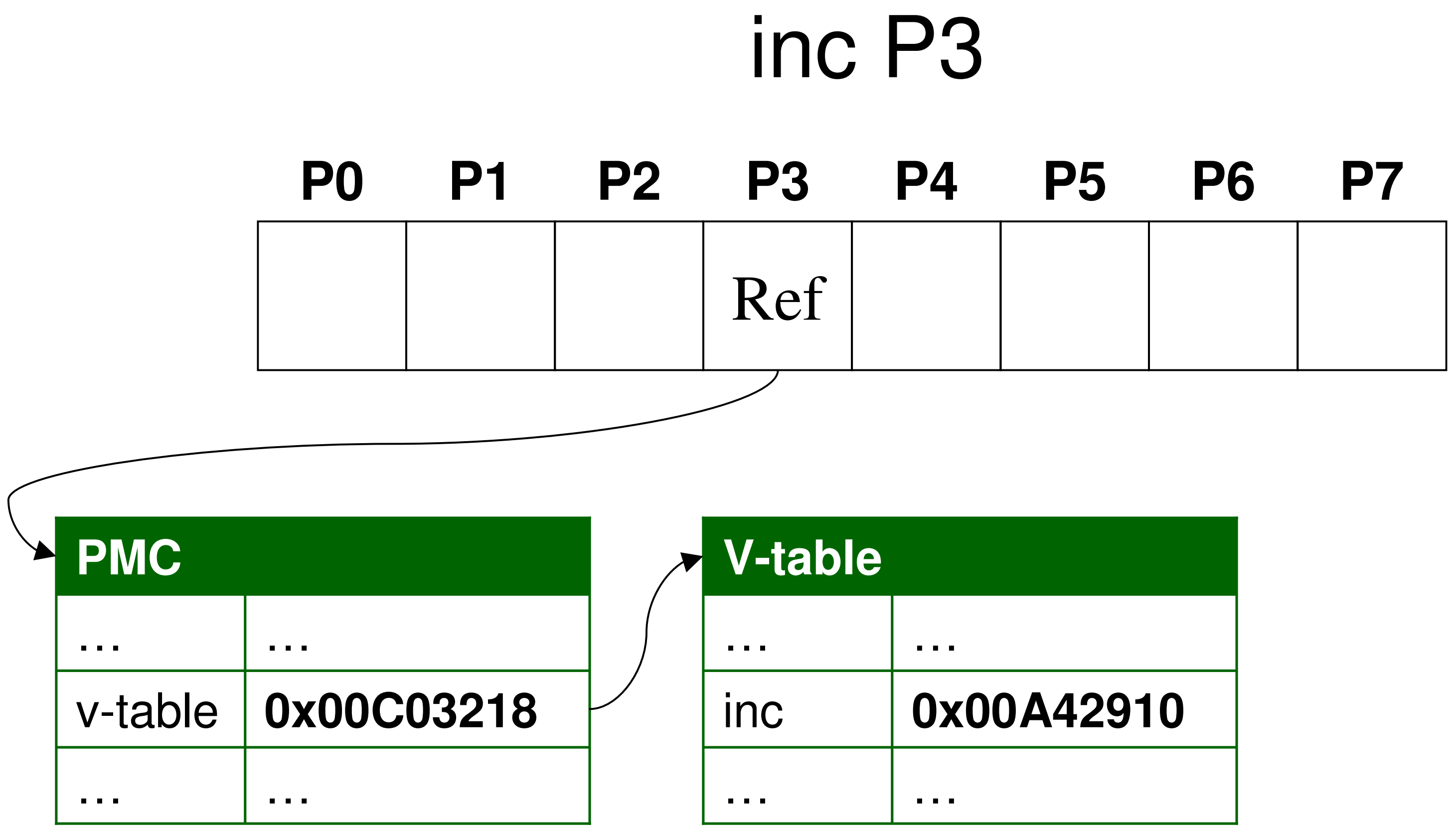
Parrot – What, where and why?

How do PMCs work?



Parrot – What, where and why?

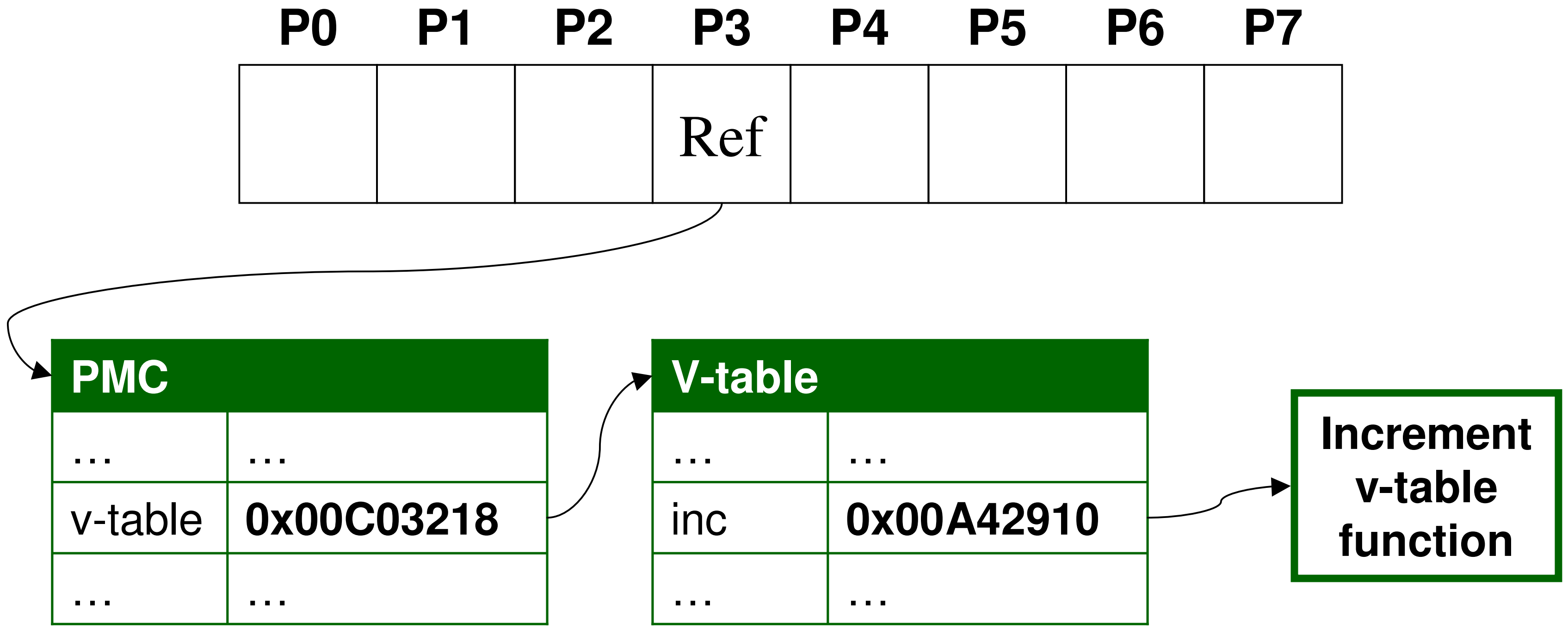
How do PMCs work?



Parrot – What, where and why?

How do PMCs work?

inc P3



Parrot – What, where and why?

PMCs allow language specific behaviour

- The same operation in two languages may produce very different behaviour.
- Consider the increment operator (++) performed on the string “ABC”.
  - In Perl, the string becomes “ABD”.
  - In Python, an exception is thrown.
- PerlString and PythonString PMCs can implement the “increment” method differently.

Parrot – What, where and why?

PMCs enable language interoperability

- PMCs not only have methods to perform operations but also to get and set the data stored in them in integer, number and string form.
- The PerlString PMC need not know the internals of another language’s string PMC.
- Simply call get_string on the other language’s PMC to get the string value as a standard Parrot string.

Parrot – What, where and why?

PMCs support aggregate types

- PMCs have v-table methods for keyed get and set (where the key is an integer, string or PMC).
- These provide an interface for implementing arrays and dictionary data structures (such as hash tables).
- Storage mechanism left for the PMC to implement (e.g. a BitArray PMC could be implemented that uses 1 bit per element).

Parrot – What, where and why?

PMCs do even more stuff!

•Provide the basis for the implementation of an object system with v-table methods such as add_parent, add_method find_method, isa and more.

•A standard way to provide access to Parrot features such as subs, coroutines and continuations.

•PMCs simultaneously solve many problems through a single simple mechanism.

Parrot – What, where and why?

Where are we at with PMCs?

•Most PMC related stuff has worked pretty solidly for a while. The PMC tool chain is pretty good.

•Dynamically loadable PMCs, stored in DLLs, currently do not work on some platforms. Support on others is a bit messy.

•More Parrot features will come to be presented as PMCs, such as I/O.

Parrot – What, where and why?

What is a run core?

- Takes Parrot bytecode and executes it.
- Involves mapping Parrot instructions to instructions supported by the hardware.
- We would like:
  - High portability
  - High performance
- These often turn out to be opposing goals.

Parrot – What, where and why?

Interpreting Parrot Bytecode

- For each Parrot instruction write code in C to perform the instruction.
- These are written in a standard format.

```
inline op add(out INT, in INT, in INT) :base_core {
  $1 = $2 + $3;
  goto NEXT();
}
```

- An build tool takes these and generates a run core by adding logic to move between instructions and execute each one.

Parrot – What, where and why?

The function call per op run cores

- The build tool generates a function for each instruction and a table of function pointers.
- Execute instructions by looking up the function pointer in the table for that instruction then calling the function.
- Possible to add profiling and bounds checking code between operations.
- Completely portable, but performance hit due to making a function call per instruction.

Parrot – What, where and why?

The switch run core

•A huge "switch" block is generated with a case for each Parrot instruction.

•After executing an instruction, the program counter is increment and we jump back to the top of the switch block again (using goto).

•Performance depends heavily on the code the compiler generates for switch blocks, but no per-op function call overhead is a bonus.

•Standard C so also completely portable.

Parrot – What, where and why?

The computed goto run core

- GCC allows goto to jump to a memory address computed at runtime rather than a named label like most other compilers.
- Emit C code for each instruction into a function, prefix it with a label and build a table of label addresses.
- After executing each instruction, look up the address of the C code for the next instruction using the table and goto that address.

Parrot – What, where and why?

The computed goto run core

•Computed goto is the highest performing interpreter run core.

•Only works on a small number of compilers, so not very portable.

•Code that uses computed goto interacts nastily with the C compiler's optimizer – basically the optimizer can't do much with it.

•Tends to mean that the computed goto core takes a lot of time and memory to compile.

Parrot – What, where and why?

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.

Parrot – What, where and why?

How does JIT work?

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

Parrot – What, where and why?

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.

Parrot – What, where and why?

Why so many run cores?

- The function-call run cores support debugging, tracing, profiling and JIT fallback.
- The switch or c-goto run cores offer good performance on platforms with no JIT.
- JIT can offer very fast execution.
    - Has compilation time overhead – research suggests short lived programs can run faster if just interpreted.

Parrot – What, where and why?

Where are the run cores at?

•All of the interpreted ones are implemented and work.

•Quite a few Parrot ops can be JIT compiled on x86, PPC and Sun4.

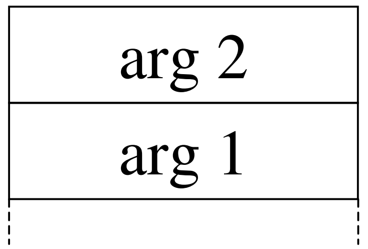
•There is limited JIT support for MIPs, Alpha, IA64 and ARM, though some of these are broken due to internals changes.

•No AOT (Ahead Of Time) compilation yet; lots of room for improvements with JIT.

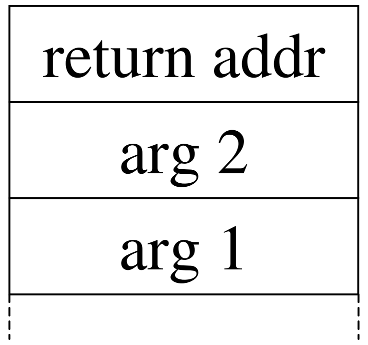
Parrot – What, where and why?

How Parrot doesn't do sub and method calls

- The traditional way to call a function involves using a stack.
- Arguments are placed on the stack.



- The program counter for the next instruction (aka return address) is put on the stack and a jump made to the function.



Parrot – What, where and why?

How Parrot doesn't do sub and method calls

- After the function has executed, the return value is placed either on the stack or in an agreed register.
- The return address is popped off the stack and jumped to, returning control to the caller.
- For deeply recursive calls, a big stack is built up. Some systems have limited stack space.
- Security issues – what if bad code allows the return address to be overwritten?

Parrot – What, where and why?

<u>Parrot uses Continuation Passing Scheme</u>

•Each instance of a sub or method in the call chain has its own set of registers that store its current working data.

•Lexicals are also stored in registers.

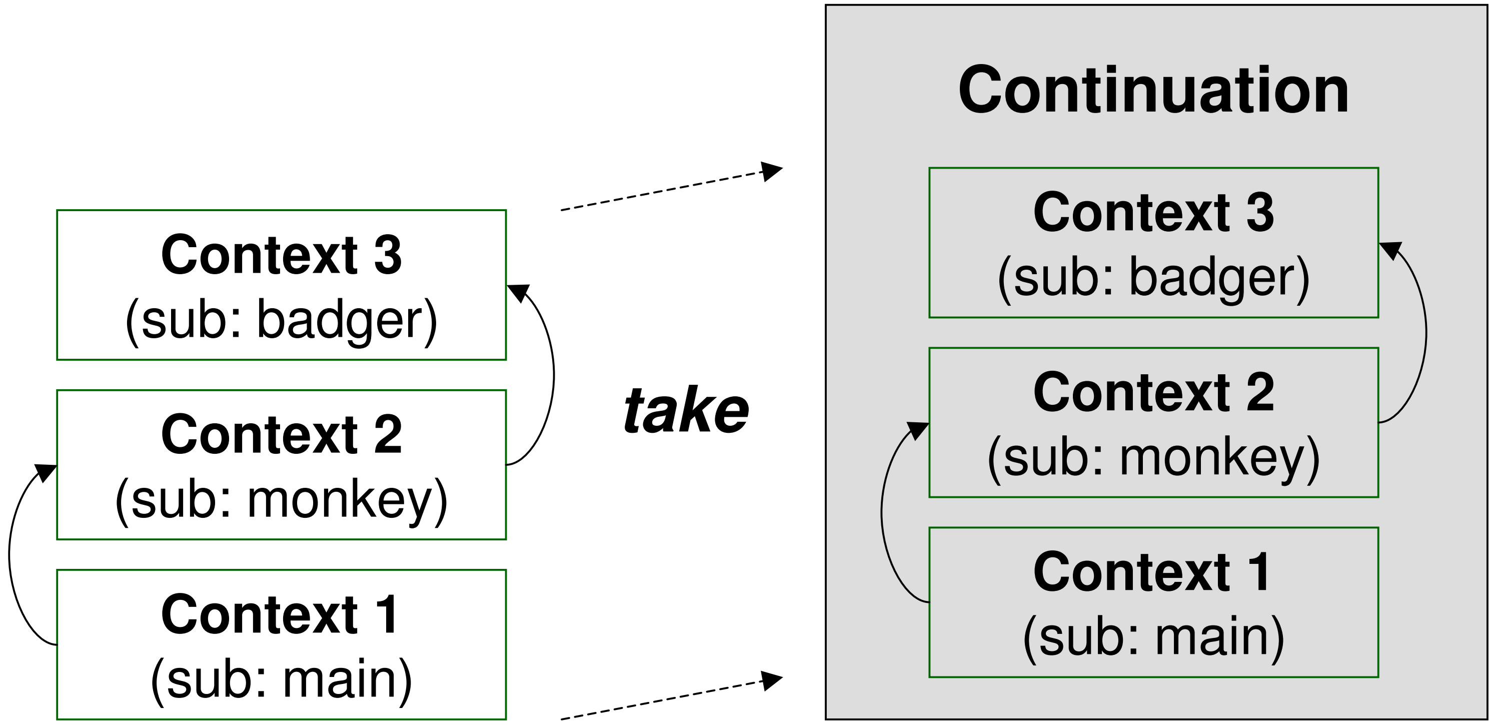
•Along with various other bits of data related to the current runtime state of a sub, these items make up a context.

•Each context points to the previous context, describing the chain of calls that was made.

Parrot – What, where and why?

Parrot uses Continuation Passing Scheme

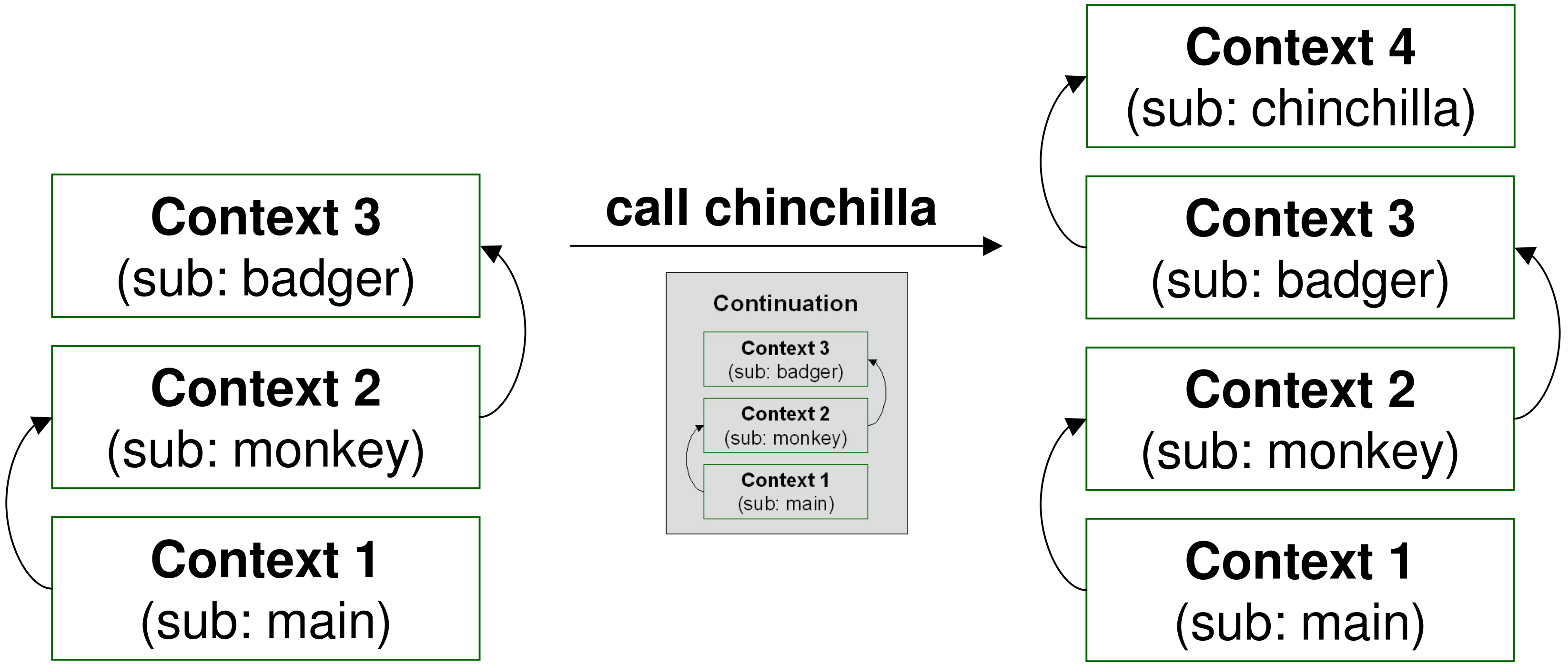
- Taking a continuation makes a copy of this chain of contexts.



Parrot – What, where and why?

Parrot uses Continuation Passing Scheme

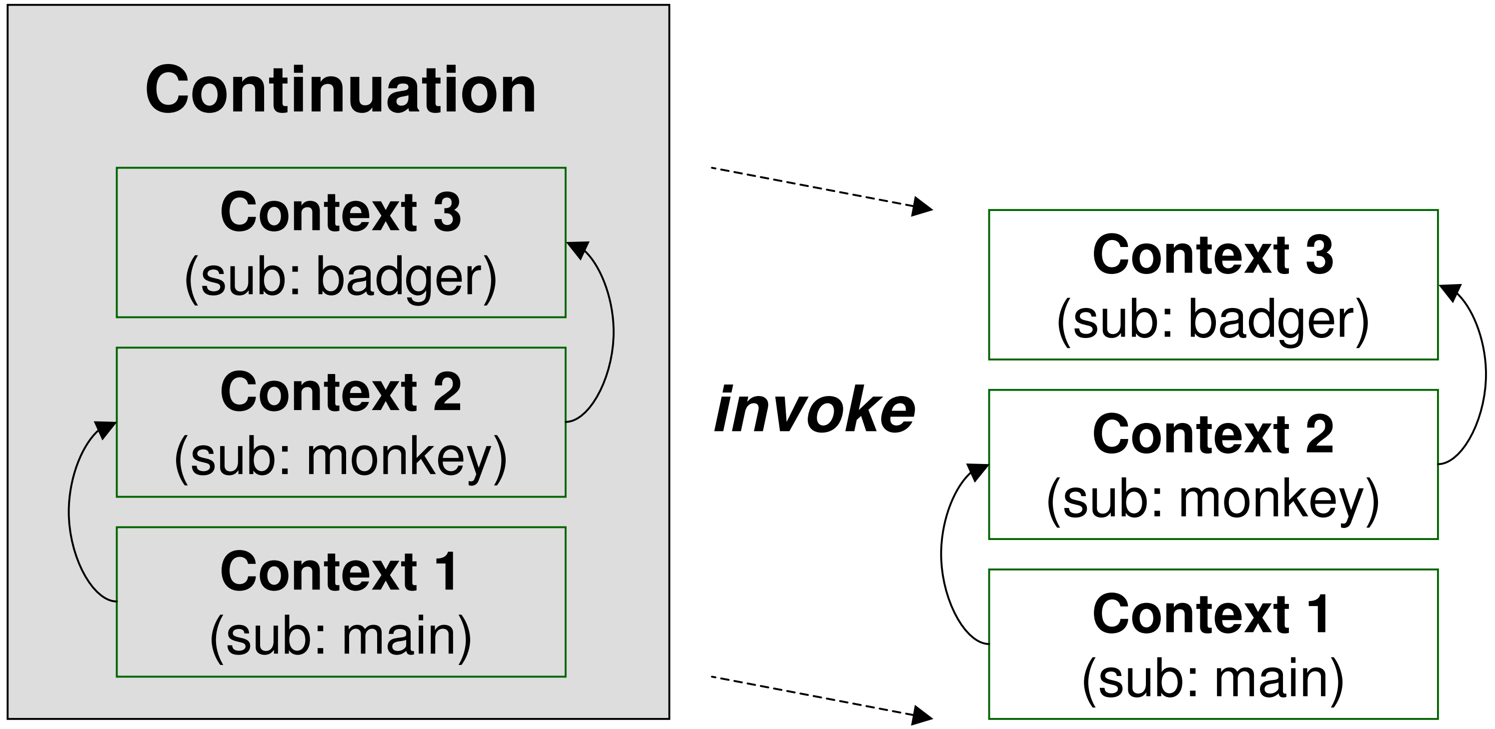
- To call, take a continuation, then jump to the sub, passing the continuation and arguments.



Parrot – What, where and why?

Parrot uses Continuation Passing Scheme

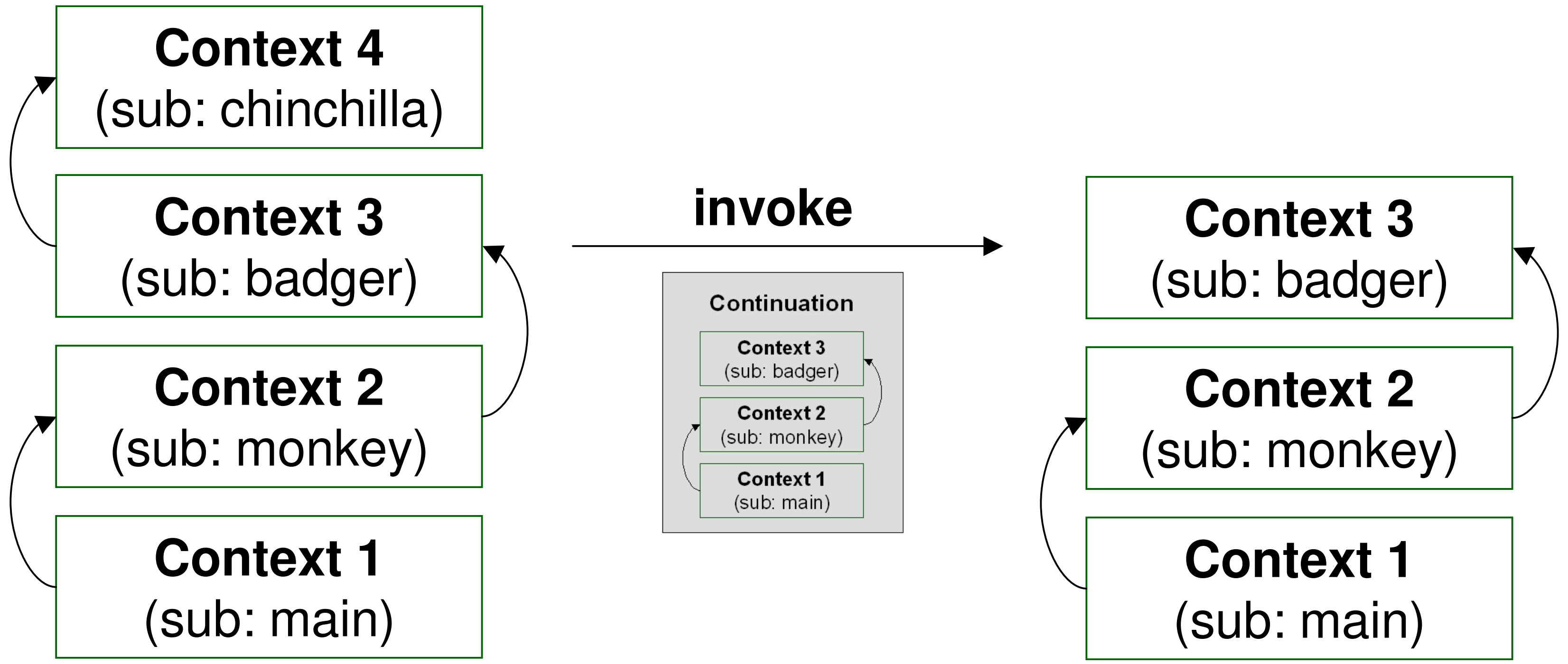
- Invoking a continuation involves replacing the current call chain with what was captured.



Parrot – What, where and why?

Parrot uses Continuation Passing Scheme

- Conveniently, this turns out to do just what a return would do!



Parrot – What, where and why?

Why Continuation Passing Scheme?

•Parrot has a lot of context information to save; continuations capture all of it neatly.

•No concerns about over-flowing the stack or over-writing return addresses.

•Sounds expensive, but can copy contexts lazily (if the return continuation becomes a full continuation), so actually quite cheap.

•Tail calls easy – just pass on the already taken return continuation.

Parrot – What, where and why?

Memory Management

•During their execution, programs allocate memory for storing working data in.

•Often this memory is only used for a short amount of time.

•There is only a finite amount of memory available to use, so programs need to free up memory that is no longer being used.

•Traditionally programs did this themselves, e.g. through malloc() and free() in C.

Parrot – What, where and why?

What is GC (Garbage Collection) and why?

- Garbage collection systems automate the freeing of memory when it is no longer in use.
- The programmer is no longer responsible for freeing memory meaning:
  - No memory leaks.
  - No chance of accidentally freeing things that are still in use.
  - Faster development.

Parrot – What, where and why?

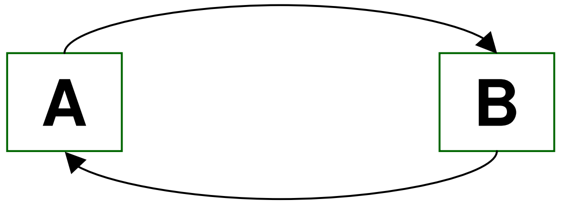
What is reference counting?

- An approach to garbage collection, used in Perl 5 but not Parrot.
- Every object has a reference count – a value that keeps track of the number of variables and other objects that refer to that object.
- When the reference count reaches zero, there is no way the object could be accessed, so it is no longer in use, therefore it can be freed.

Parrot – What, where and why?

Why Parrot isn't using reference counting

- Very easy to forget to increment or decrement the reference count as needed.
- Garbage collection complexity spread across the entire code base.
- Circular data structures never get freed as their reference count never reaches zero.


$$\begin{tikzcd} \boxed{A} \arrow[r, bend left] & \boxed{B} \arrow[l, bend left] \end{tikzcd}$$

Parrot – What, where and why?

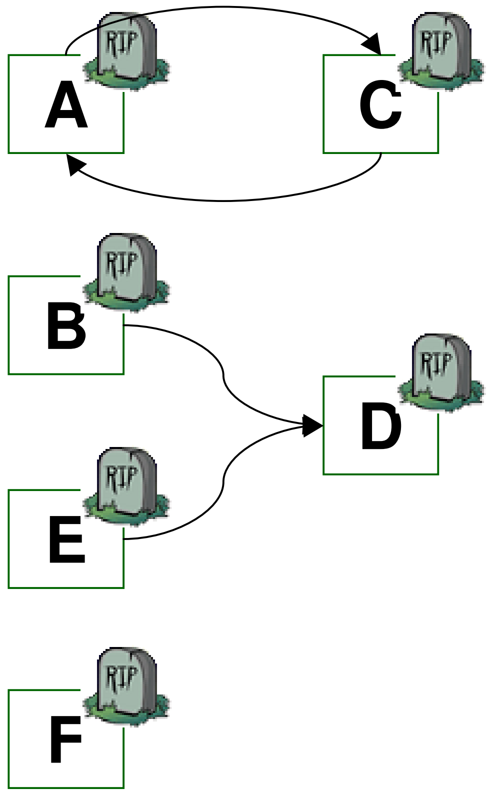
How does Parrot do GC?

- Parrot knows the locations of all objects that are eligible for GC (PMCs and strings).
    - These are allocated out of memory pools.
- GC runs when all memory in the pools is allocated to see if some can be freed rather than growing the pool or when the program requests it to (and maybe in some other cases).
- Split up into two steps: DOD and sweep.

Parrot – What, where and why?

Dead Object Detection (DOD)

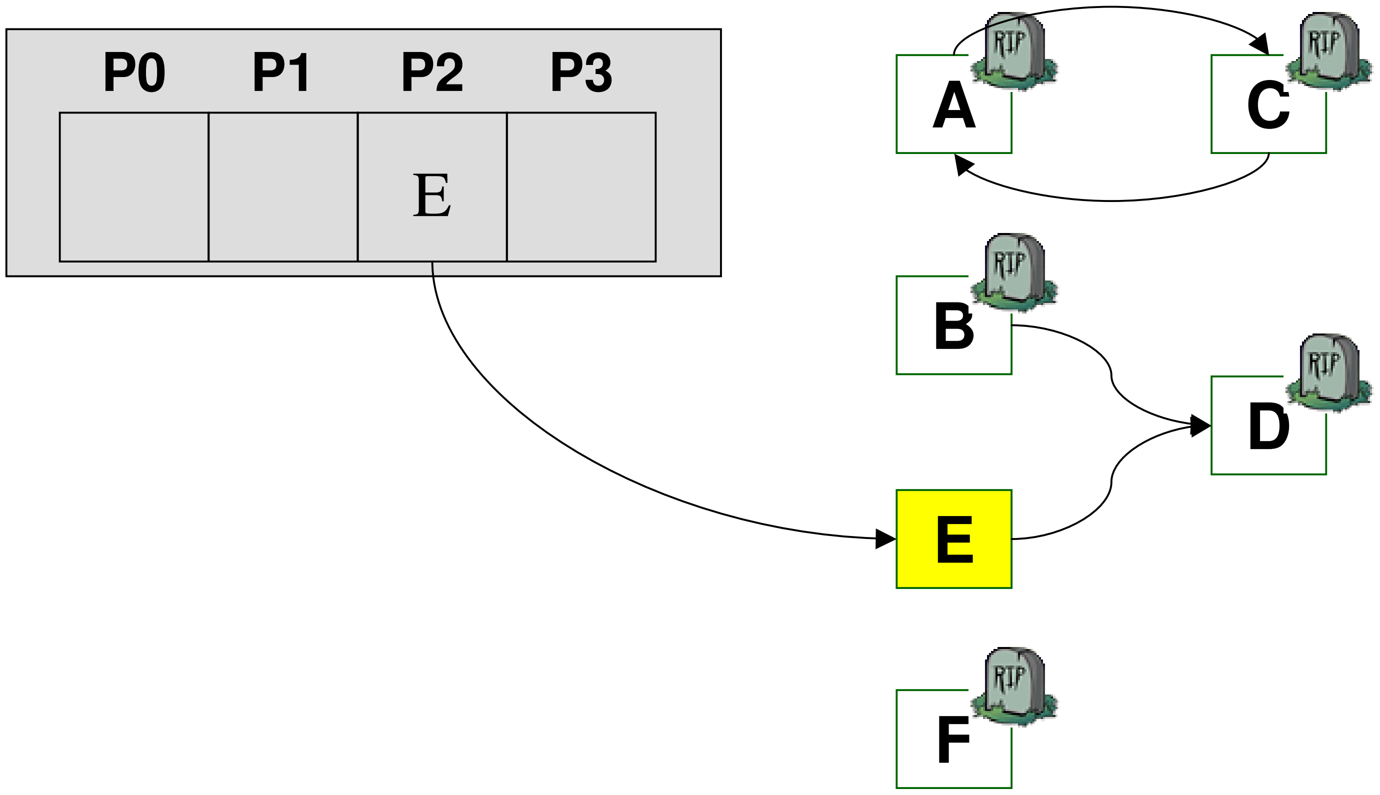
- Initially consider all objects dead (that is, unreachable).



Parrot – What, where and why?

Dead Object Detection (DOD)

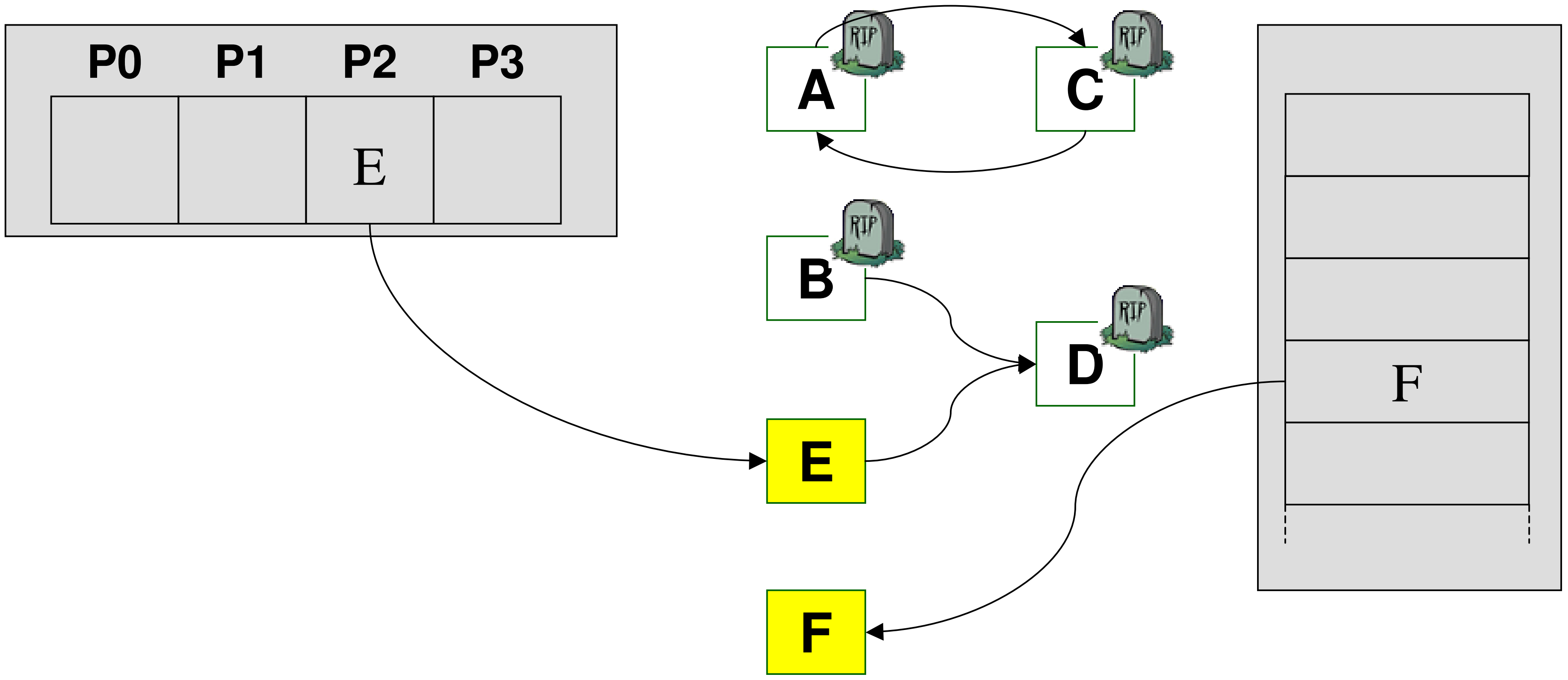
•Mark any objects that are referenced from Parrot registers as alive.



Parrot – What, where and why?

Dead Object Detection (DOD)

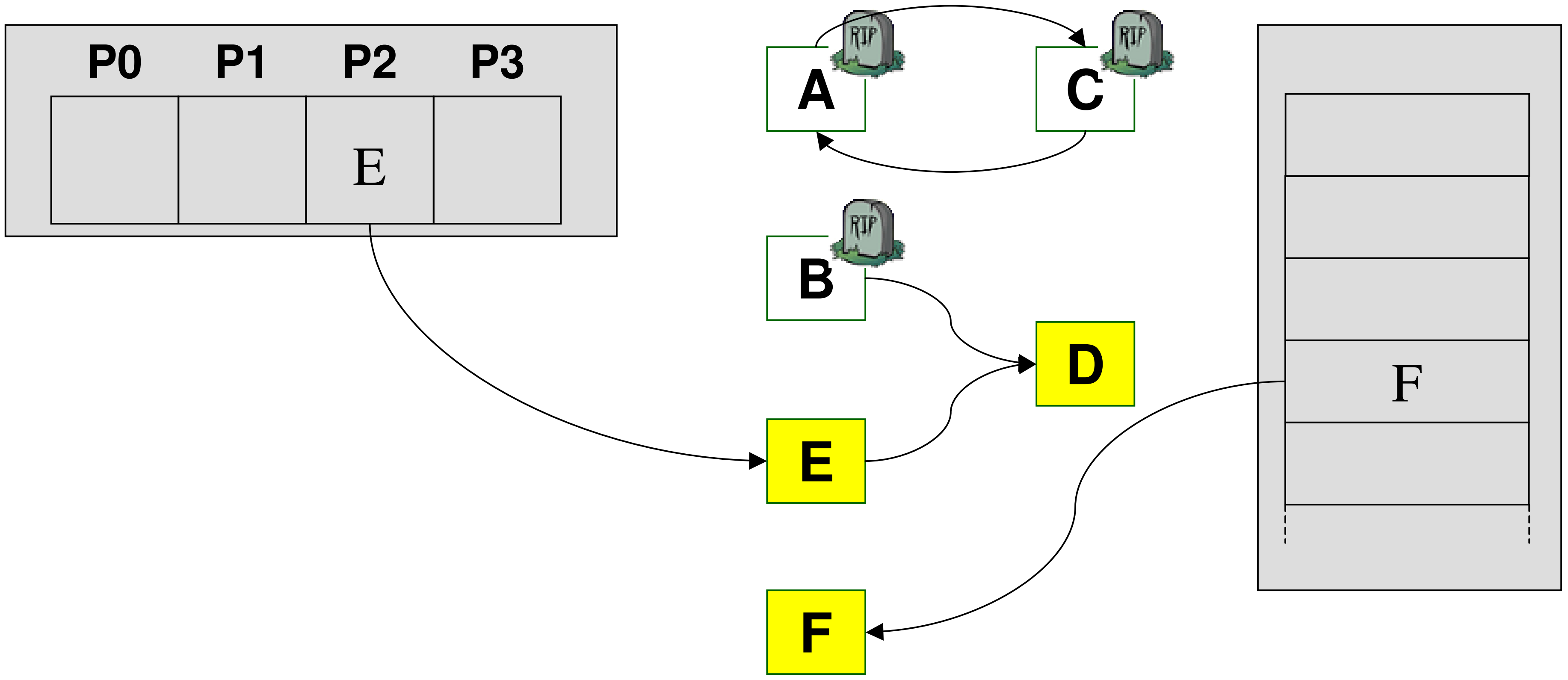
- Look at the system stack for the Parrot VM and mark referenced objects alive.



Parrot – What, where and why?

Dead Object Detection (DOD)

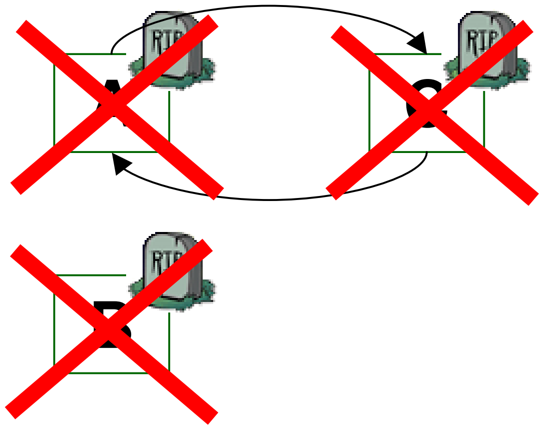
- Finally, transitively mark objects referenced by live objects as alive.



Parrot – What, where and why?

Sweep

- Objects that were not marked alive can thus have the memory associated with them freed.



- Finalizers (program level clean-up) and destructors (VM level clean-up) will be called before the object’s memory is freed.

Parrot – What, where and why?

Why does Parrot do GC this way?

- Complexity of GC contained in a small part of the code base, not spread throughout it, thus simpler to debug and smaller code.
- Better performance – no ref counts to ++/--
- Circular data structures no longer a problem.
- Separate DOD and sweep stages aid multi-threading performance – sweep unlikely to need any locks.

Parrot – What, where and why?

Where is Parrot’s GC at?

•It works!

•New bugs in the GC system occasionally discovered but for the most part it’s stable.

•Generational and incremental GC schemes have been implemented, though are not used in a default Parrot build.

•A thread aware GC has been implemented but is in a branch and is so far unused.

Parrot – What, where and why?

How will Parrot support concurrency?

- Threads will be implemented using the operating system's thread support.
    - The OS can schedule threads on multiple CPUs, which will be really important soon.
- Concurrency control with STM (Software Transactional Memory).
    - Like transactions in databases, but much more lightweight; STM is highly scalable and provides a good programmer model.

Parrot – What, where and why?

Where is Parrot's concurrency support at?

•Threads are implemented on a number of platforms and basically work.

•Parrot threads are reported to be much more lightweight than Perl 5's ithreads.

•STM not implemented at all in Parrot yet, but it is in The Plans. Currently some more primitive locking mechanisms are in place.

•The specification for concurrency needs an overhaul and updating to account for STM.

Parrot – What, where and why?

Other things that need work include…

- The I/O subsystem will be presented as a number of PMCs, but at the moment many operations are Parrot instructions and some things are very likely just not implemented.
- Events and asynchronous I/O need to be fully specified and implemented.
- There is a specification for the security model, but it is marked as a draft and not implemented yet.

Parrot – What, where and why?

Other things that need work include…

- The Parrot compiler tool chain; the Parrot Grammar Engine is coming along well, and a Tree Transformation Engine is in the works. A preliminary Parrot AST is implemented.
- Finalising the specification and implementation of namespaces and exceptions and objects .
- Character set support is coming along, but there’s more to do.

Parrot – What, where and why?

Conclusion

- Parrot can do a lot already.
- Equally, Parrot still has some way to go.
- Parrot is innovative and not just a .NET or JVM clone.
- Parrot will make things better for Perl users.
- Parrot is fun!
- Any questions?