

The Perl Compiler & Development Environment

Malcolm Beattie

Oxford University Computing Services

O'Reilly Perl Conference, San Jose, 1997

20 Aug 1997

Overview

- The compiler
 - how it's used
 - what can/can't/will be able to be compiled
 - backends
 - implementation
- Multi-threading
 - programming (threads, locks, condition variables, queues)
 - implementation
- Optional strong typing (my Dog \$spot)
- Availability and Release Schedule

The compiler

- how it's used
- what can/can't/will be able to be compiled
 - simple programs, programs with complications, programs that use .pm modules, programs that use simple XSUB modules, programs that use complex XSUB modules, .pm modules, XSUB modules
- backends
 - C, CC, Bytecode, Xref, Lint, ...
- implementation
 - OPs and SVs, the B class, walkoptree

How to use the compiler

- For an ordinary program, a typical invocation of the compiler is

```
perl -MO=CC, -ofoo.c foo.pl  
perl cc_harness -o foo foo.c
```

- In a future release, this will become simply

```
perl -MO=CC, -ofoo foo.pl
```

- Compiling a .pm module into an XSUB module is currently possible but is platform-dependent and inelegant.
- The aim is for a simple user interface but some of the more primitive platforms may make this impossible.

What will the compiler compile?

- Currently, it compiles programs that...
 - ✓ use simple, plain, ordinary constructs
 - ✓ use simple .pm modules
 - ✗ don't use the DATA filehandle, operator overloading, goto or global end-of-program destructors
 - ✗ don't use a few extremely weird and wacky perl "features"
 - ✗ don't use complex XSUB modules
- With manual assistance it may compile programs that...
 - ✓ use simple XSUB modules (modulo link/library issues)
- Forthcoming versions can compile .pm and XSUB modules

Backends

- The **C** backend walks the internal parse tree and writes out a C program which encodes all the relevant structures.
- The **CC** backend walks the parse tree in execution order and translates it into linear C code. Basic block/stack/lexical variable analysis turn some Perl OPs into inline C code.
- The **Bytecode** backend writes out platform-independent bytecode for saving/restoring the parse tree (like **C**, not **CC**).
- The **Xref** backend generates a cross-reference report of filenames and line numbers of a program's variables and subs.
- The **Lint** backend walks the parse tree of a program and warns about possible problems. Currently it can only warn about implicit scalar context and implicit use of \$_.

Compiler implementation

- `perl -MO=Foo, a, b, c ...` invokes `use 0 qw(Foo a b c)` which establishes an `END{}` hook to call `B::Foo::compile`.
- Perl's internal "object" structures (OPs, SVs and derivatives) have corresponding classes `B::OP`, `B::UNOP`, `B::SV`, ...
 - methods for each structure field return the data
 - `f = op->op_flags` in C becomes `$f = $op->f_flags` in Perl
 - fields holding OP and SV pointers in C become Perl objects blessed into the appropriate classes
- `walkoptree($rootop, "foo")` walks the parse tree rooted at `$rootop` calling method `foo` on each OP object
- **CC** is more sophisticated

Multi-threading

- programming
 - threads: creating, joining, detaching
 - locks: block scoped auto-unlocking recursive mutexes
 - condition variables
 - higher level modules: queues, events, worker pools, ...
- differences from non-threaded perl
 - lexical `@_`, `fork()`/`backticks/system()`, signals
- implementation
 - `struct thread`, `thr->Tfoo`, `dTHR`, `MUTEX_*`, `COND_*`, per-thread lexicals, O/S threads desirable.

Programming: threads

- Creating and joining with threads

```
sub start_here {  
    my ($foo, $bar, $baz) = @_;  
    ...  
    return @results;  
}  
  
$t = new Thread \&start_here, 1, 2, 3;  
...  
@answers = $t->join;
```

- Detach threads with `$t->detach`;
- subs can be synchronised (cf. Java)

Programming: locks

- `lock($q)`; waits until it gains a lock on `$q`
- Automatically attaches a lock to any scalar
- `lock()` of a scalar reference locks the referenced object:
`$a = new Obj; $b = $a;`
`lock($a)` and `lock($b)` operate on the same lock.
- The lock is *recursive*: the same thread can immediately gain a lock that it already holds (and not deadlock).
- Unlocking happens automatically on leaving the enclosing block—even via `return()` or `die()`.
- Higher level semantics are layered on top of locks.

Programming: condition variables

- `cond_wait($q)`, `cond_signal($q)`, `cond_broadcast($q)`
- Automatically attaches a condition variable to any scalar
- Use `lock()` on the same variable to protect the condition

```
sub enqueue {  
    my $q = lock shift;  
    push(@$q, @_);  
    cond_signal($q) if @$q; }  
  
sub dequeue {  
    my $q = lock shift;  
    cond_wait($q) while @$q == 0;  
    return shift @$q; }
```

Programming: higher level

- Queues—see example on “condition variables” slide
- Events can basically use an ordinary queue
 - `$q->enqueue($event)` enqueues a new event
 - `$event = $q->dequeue` gets the next event
- Worker pools can contend for an ordinary queue of tasks

```
$tq = new Thread::Queue;
sub worker {
    while (1) { &{$tq->dequeue} }
}
for ($i = 0; $i < $n; $i++) { async(\&worker) }
$tq->enqueue(\&task);
```

Differences from non-threaded Perl

- Lexical @_
- fork() / backticks / system()
- signal handling
- synchronise access to globals and shared variables

Multi-threaded perl: implementation

- `struct thread`
some old global fields + some old per-interpreter fields + some new fields
- Global struct thread *thr
 - `#define foo thr->Tfoo`
 - thr passed as argument to PP code
 - dTHR gets thr from thread-specific data
- `MUTEX_*` and `COND_*` macros for simple synchronisation API
- Core changes: `per-thread` lexicals, `pp_lock`, `pp_enter` sub
- Underlying threads via `POSIX.1b` or “-DFAKE_THREADS” API

Optional Strong Typing

- Never mandatory
- Used for better optimisation and warnings
- `my Dog $spot` tags a *lexical* variable with a class name
- Some class names pre-defined for use by perl (e.g. `int`, `num`)
- Pseudo-hashes: can treat an array reference as a hash reference
 - `$a->{foo}` looks for `$a->[0]->{foo}`
 - A non-existent key is an error
 - A found key, *i*, makes `$a->{foo}` become `$a->[i]`
 - If `%Dog::FIELDS` exists then `$spot->{foo}` uses it to look up the index at compile-time

Availability and Release Schedule

- Development releases available from August
- Development freeze around October
- Perl 5.005 in November
 - integrate from maintenance release
 - can build with or without thread support
 - thread support will not be mature
 - compiler support (not very noticeable at programmer level)
 - some strong typing allowed for compiler and warning hints