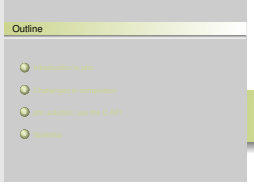
A Practical Solution for Scripting Language Compilers
Paul Biggar, Edsko de Vries and David Gregg
Department of Computer Science and Statistics
Trinity College Dublin
SAC '09: 11th March, 2009

A Practical Solution for Scripting Language Compilers

Paul Biggar, Edsko de Vries and David Gregg

Department of Computer Science and Statistics
Trinity College Dublin

SAC '09: 11th March, 2009

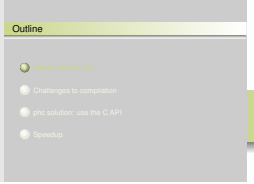


Outline

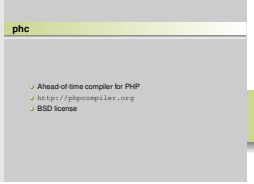
Sneak peak
Problem: Scripting languages present "unique" problems (in practice)
Solution: Re-use as much of the *Canonical Implementation* as possible.

Sneak peak

- Problem: Scripting languages present "unique" problems (in practice)
- Solution: Re-use as much of the *Canonical Implementation* as possible.



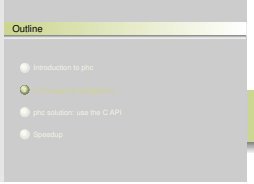
Outline



phc

1. BSD licence useful since its easy to extend

- Ahead-of-time compiler for PHP
- `http://phpcompiler.org`
- BSD license



Outline

Undefined Language Semantics

The PHP group claim that they have the final say in the specification of PHP. This group's specification is an implementation, and there is no prose specification or agreed validation suite. There are alternate implementations [...] that claim to be compatible (they don't say what this means) with some version of PHP.

D. M. Jones. Forms of language specification: Examples from commonly used computer languages. ISO/IEC JTC1/SC22/OWG/N0121, February 2008.

Undefined Language Semantics

The PHP group claim that they have the final say in the specification of PHP. This group's specification is an implementation, and there is no prose specification or agreed validation suite. There are alternate implementations [...] that claim to be compatible (they don't say what this means) with some version of PHP.

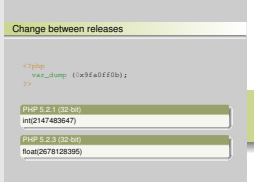
D. M. Jones. Forms of language specification: Examples from commonly used computer languages. ISO/IEC JTC1/SC22/OWG/N0121, February 2008.

1. all written in C, not PHP
2. Mike Furr earlier: 1000 methods/classes in C
3. 4870 functions, 1000 methods

Batteries included

abs()
acos()
acosh()
addcslashes()
addslashes()
aggregate()
aggregate_info()
aggregate_methods()
aggregate_methods_by_list()
aggregate_methods_by_regexp()
aggregate_properties()
aggregate_properties_by_list()
aggregate_properties_by_regexp()
aggregation_info()
apache_child_terminate()
apache_get_modules()
apache_get_version()
apache_getenv()
apache_lookup_uri()
apache_note()
apache_request_headers()
apache_reset_timeout()
apache_response_headers()
apache_setenv()
apc_add()
apc_cache_info()
apc_clear_cache()
apc_compile_file()
apc_define_constants()
apc_delete()
apc_fetch()
apc_load_constants()
apc_sma_info()
apc_store()
apd_breakpoint()
apd_callstack()
apd_clunk()
apd_continue()
apd_croak()
apd_dump_function_table()
apd_dump_persistent_resources()
apd_dump_regular_resources()
apd_echo()
apd_get_active_symbols()
apd_set_pprof_trace()
apd_set_session()
apd_set_session_trace()
apd_set_socket_session_trace()
array()
array_change_key_case()
array_chunk()
array_combine()
array_count_values()
array_diff()
array_diff_assoc()
array_diff_key()
array_diff_uassoc()
array_diff_ukey()
array_fill()
array_fill_keys()
array_filter()
array_flip()
array_intersect()
array_intersect_assoc()
array_intersect_key()
array_intersect_uassoc()
array_intersect_ukey()
array_key_exists()
array_keys()
array_map()
array_merge()
array_merge_recursive()
array_multisort()
array_pad()
array_pop()
array_product()
array_push()
array_rand()
array_reduce()
array_reverse()
array_search()
array_shift()
array_slice()
array_splice()
array_sum()
array_udiff()
array_udiff_assoc()
array_udiff_uassoc()
array_uintersect()
array_uintersect_assoc()
array_uintersect_uassoc()
array_unique()
array_unshift()
array_values()
array_walk()
array_walk_recursive()
ArrayIterator::current()
ArrayIterator::key()
ArrayIterator::next()
ArrayIterator::rewind()
ArrayIterator::seek()
ArrayIterator::valid()
ArrayObject::__construct()
ArrayObject::append()
ArrayObject::count()
ArrayObject::getIterator()
ArrayObject::offsetExists()
ArrayObject::offsetGet()
ArrayObject::offsetSet()
ArrayObject::offsetUnset()
arsort()
ascii2ebcdic()
asin()
asinh()
asort()
aspell_check()
aspell_check_raw()
aspell_new()
aspell_suggest()
assert()
assert_options()
atan()
atan2()
atanh()

Jeff Atwood, Coding Horror, May 20th, 2008
http://www.codinghorror.com/blog/archives/001119.html



Change between releases

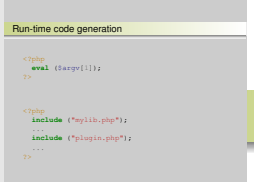
```
<?php
  var_dump (0x9fa0ff0b);
?>
```

PHP 5.2.1 (32-bit)

int(2147483647)

PHP 5.2.3 (32-bit)

float(2678128395)

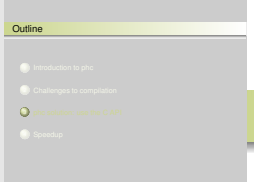


Run-time code generation

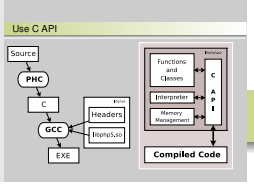
1. scripting langs are typically made for interpreters
2. can do source inclusion at compile time
3. same mechanism for plugins

```php
<?php
  eval ($argv[1]);
?>
```

```php
<?php
  include ("mylib.php");
  ...
  include ("plugin.php");
  ...
?>
```

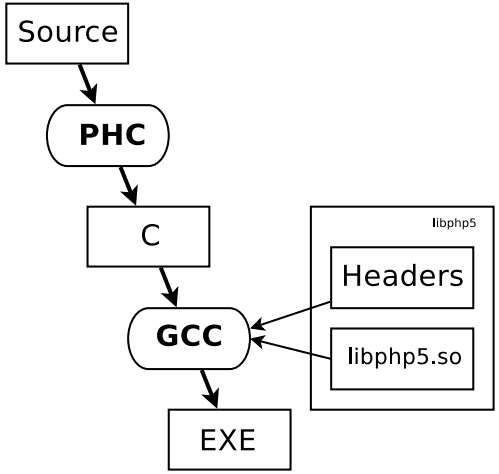


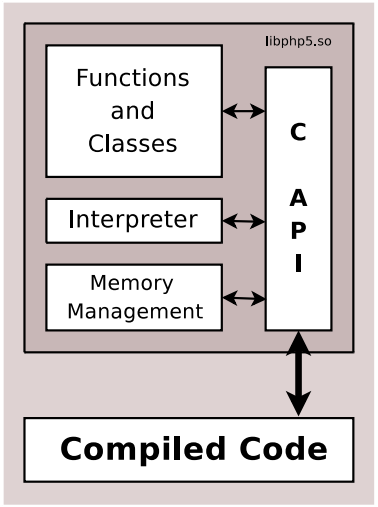
Outline



Use C API

1. RTCG
2. Functions
3. Changes between releases: also use C API at compile-time





More detail

PHP	zval
Python	PyObject
Ruby	VALUE
Lua	TValue

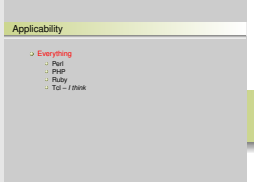
H. Muhammad and R. Ierusalimschy. C APIs in extension and extensible languages. Journal of Universal Computer Science, 13(6):839–853, 2007.

More detail

1. C API is just zval + macros and functions
2. Use (target) PHP's C API at run-time

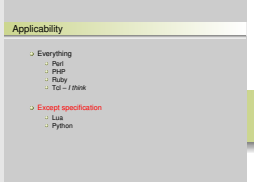
PHP	zval
Python	PyObject
Ruby	VALUE
Lua	TValue

H. Muhammad and R. Ierusalimschy. C APIs in extension and extensible languages. Journal of Universal Computer Science, 13(6):839–853, 2007.



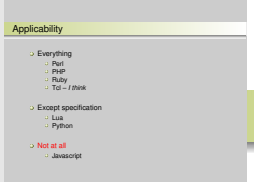
Applicability

- Everything
  - Perl
  - PHP
  - Ruby
  - Tcl – *I think*



Applicability

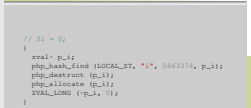
- Everything
  - Perl
  - PHP
  - Ruby
  - Tcl – *I think*
- Except specification
  - Lua
  - Python



Applicability

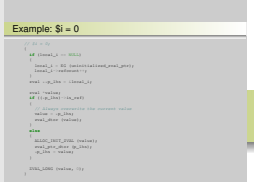
- Everything
  - Perl
  - PHP
  - Ruby
  - Tcl – *I think*
- Except specification
  - Lua
  - Python
- Not at all
  - Javascript

Simple listings: $i = 0



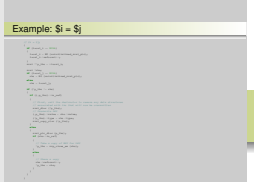
Simple listings: $i = 0

```
// $i = 0;
{
  zval* p_i;
  php_hash_find (LOCAL_ST, "i", 5863374, p_i);
  php_destruct (p_i);
  php_allocate (p_i);
  ZVAL_LONG (*p_i, 0);
}
```



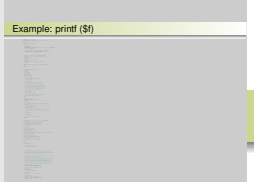
Example: $i = 0

```
// $i = 0;
{
  if (local_i == NULL)
  {
    local_i = EG (uninitialized_zval_ptr);
    local_i->refcount++;
  }
  zval **p_lhs = &local_i;

  zval *value;
  if ((*p_lhs)->is_ref)
  {
    // Always overwrite the current value
    value = *p_lhs;
    zval_dtor (value);
  }
  else
  {
    ALLOC_INIT_ZVAL (value);
    zval_ptr_dtor (p_lhs);
    *p_lhs = value;
  }

  ZVAL_LONG (value, 0);
}
```

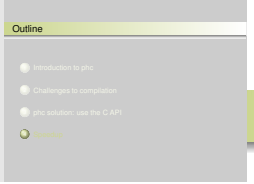


Example: $i = $j

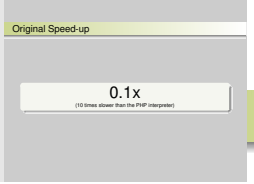
```
// $i = $j;
{
  if (local_i == NULL)
  {
    local_i = EG (uninitialized_zval_ptr);
    local_i->refcount++;
  }
  zval **p_lhs = &local_i;

  zval *rhs;
  if (local_j == NULL)
    rhs = EG (uninitialized_zval_ptr);
  else
    rhs = local_j;

  if (*p_lhs != rhs)
  {
    if ((*p_lhs)->is_ref)
    {
      // First, call the destructor to remove any data structures
      // associated with lhs that will now be overwritten
      zval_dtor (*p_lhs);
      // Overwrite LHS
      (*p_lhs)->value = rhs->value;
      (*p_lhs)->type = rhs->type;
      zval_copy_ctor (*p_lhs);
    }
    else
    {
      zval_ptr_dtor (p_lhs);
      if (rhs->is_ref)
      {
        // Take a copy of RHS for LHS
        *p_lhs = zvp_clone_ex (rhs);
      }
      else
      {
        // Share a copy
        rhs->refcount++;
        *p_lhs = rhs;
      }
    }
```



Example: printf ($f)



Outline

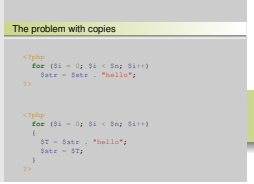


Original Speed-up

1. Why is experiemental evaluation a speedup?
2. That's an interesting result. Shouldnt compilers always be faster!!!
3. PHP's interpreter isnt slowed by interpreter loop. Rather its the level of dynamicism.

0.1x

(10 times slower than the PHP interpreter)

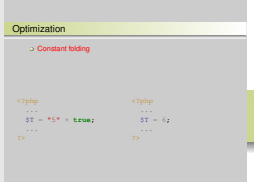


The problem with copies

1. each statement is pretty high level

```
<?php
  for ($i = 0; $i < $n; $i++)
    $str = $str . "hello";
?>
```

```
<?php
  for ($i = 0; $i < $n; $i++)
  {
    $T = $str . "hello";
    $str = $T;
  }
?>
```



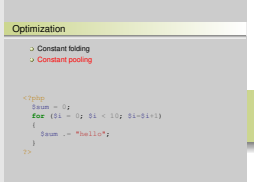
Optimization

- Constant folding

```
<?php
  ...
  $T = "5" + true;
  ...
?>
```

```
<?php
  ...
  $T = 6;
  ...
?>
```

1. We dont need to know how to fold constants - we just pass it off to PHP's eval
2. PHP implements this
3. function cant change afte first invocation - dont need lookup-cache of inline cache or polymorphic inline cache



1. We dont need to know how to fold constants - we just pass it off to PHP's eval
2. PHP implements this
3. function cant change afte first invocation - dont need lookup-cache of inline cache or polymorphic inline cache

Optimization

- Constant folding
- Constant pooling

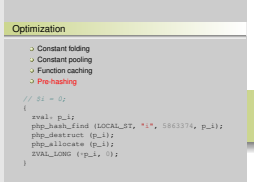
```
<?php
  $sum = 0;
  for ($i = 0; $i < 10; $i=$i+1)
  {
    $sum .= "hello";
  }
?>
```

1. We dont need to know how to fold constants - we just pass it off to PHP's eval
2. PHP implements this
3. function cant change afte first invocation - dont need lookup-cache of inline cache or polymorphic inline cache

Optimization

- Constant folding
- Constant pooling
- Function caching

```
// printf ($f);
static php_fcall_info printf_info;
{
  php_fcall_info_init ("printf", &printf_info);

  php_hash_find (
    LOCAL_ST, "f", 5863275, &printf_info.params);

  php_call_function (&printf_info);
}
```

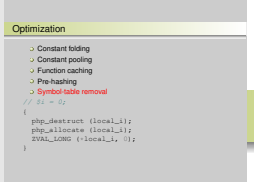


1. We dont need to know how to fold constants - we just pass it off to PHP's eval
2. PHP implements this
3. function cant change afte first invocation - dont need lookup-cache of inline cache or polymorphic inline cache

Optimization

- Constant folding
- Constant pooling
- Function caching
- Pre-hashing

```
// $i = 0;
{
  zval* p_i;
  php_hash_find (LOCAL_ST, "i", 5863374, p_i);
  php_destruct (p_i);
  php_allocate (p_i);
  ZVAL_LONG (*p_i, 0);
}
```

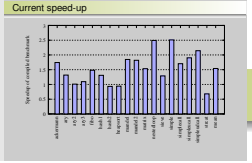


1. We dont need to know how to fold constants - we just pass it off to PHP's eval
2. PHP implements this
3. function cant change afte first invocation - dont need lookup-cache of inline cache or polymorphic inline cache

Optimization

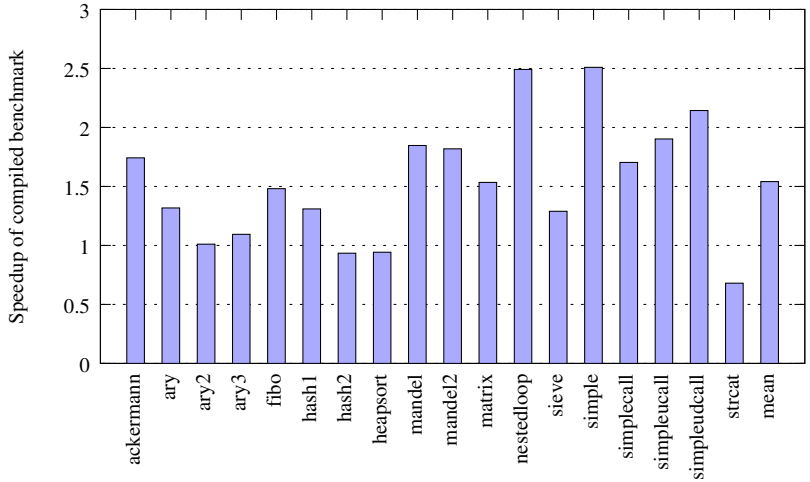
- Constant folding
- Constant pooling
- Function caching
- Pre-hashing
- Symbol-table removal

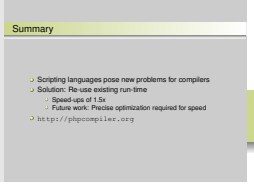
```
// $i = 0;
{
  php_destruct (local_i);
  php_allocate (local_i);
  ZVAL_LONG (*local_i, 0);
}
```



Current speed-up

1. Explain how to read graph
2. Much better than 0.1x
3. C compiler: be 5x faster
4. PHP 40x-70x slower





Summary

- Scripting languages pose new problems for compilers
- Solution: Re-use existing run-time
  - Speed-ups of 1.5x
  - Future work: Precise optimization required for speed
- `http://phpcompiler.org`