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.

Outline

phc

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

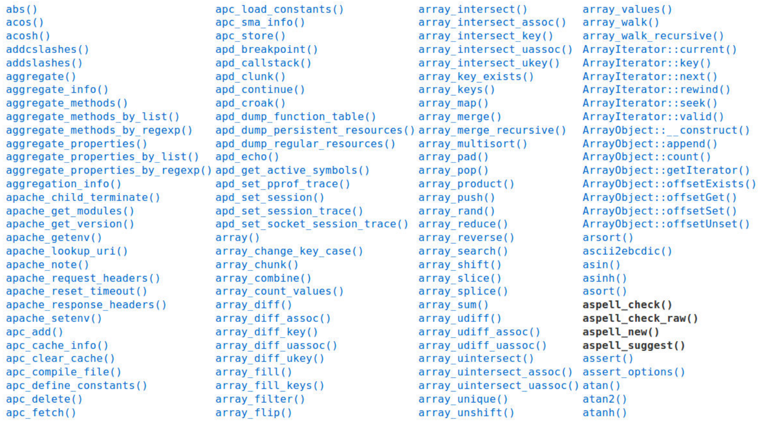
Outline

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.

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)
```

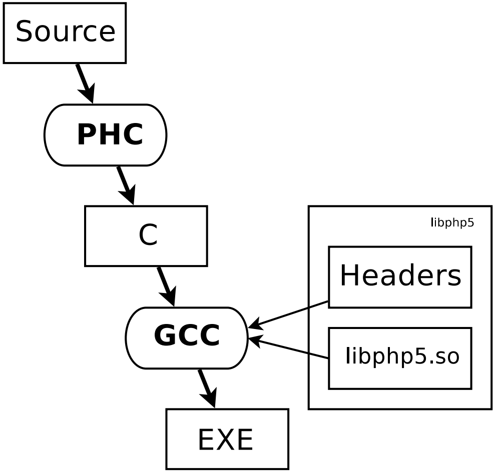
Run-time code generation

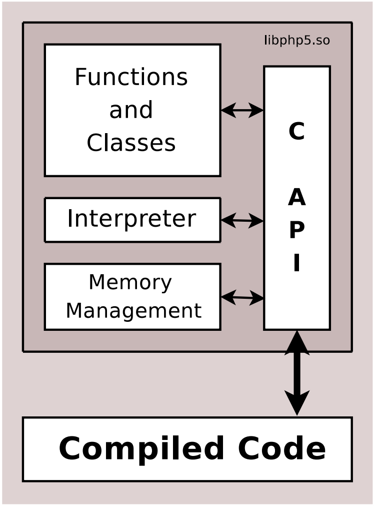
```
<?php
  eval ($argv[1]);
?>
```

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

Outline

Use C API


Source
PHC
C
GCC
EXE
libphp5
Headers
libphp5.so


libphp5.so
Functions and Classes
Interpreter
Memory Management
C API
Compiled Code

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.

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

```
// $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)

```c
static zend_fcall_info printf_fci;
static zend_fcall_info_cache printf_fcic = { 0, NULL, NULL, NULL };

// printf($f);
{
  if (!printf_fcic->initialized)
  {
    zval fn;
    INIT_PZVAL (&fn);
    ZVAL_STRING (&fn, "printf", 0);
    int result = zend_fcall_info_init (&fn, &printf_fci, &printf_fcic TSRMLS_CC);
    if (result != SUCCESS)
    {
      phc_setup_error (1, "listings_source.php", 8, NULL TSRMLS_CC);
      php_error_docref (NULL TSRMLS_CC, E_ERROR,
          "Call to undefined function %s()", function_name);
    }
  }

  zend_function *signature = printf_fcic.function_handler;
  zend_arg_info *arg_info = signature->common.arg_info; // optional

  int by_ref[1];
  int abs_index = 0;
  // TODO: find names to replace index
  if (arg_info)
  {
    by_ref[abs_index] = arg_info->pass_by_reference;
    arg_info++;
  }
  else
    by_ref[abs_index] = signature->common.pass_rest_by_reference;

  abs_index++;


  // Setup array of arguments
  // TODO: I think arrays of size 0 is an error
  int destruct[1];
  zval *args[1];
  zval **args_ind[1];

  int af_index = 0;
  destruct[af_index] = 0;
  if (by_ref[af_index])
  {
    if (local_f == NULL)
    {
      local_f = EG (uninitialized_zval_ptr);
      local_f->refcount++;
    }
    zval **p_arg = &local_f;

    // We don't need to restore ->is_ref afterwards,
    // because the called function will reduce the
    // refcount of arg on return, and will reset is_ref to
    // 0 when refcount drops to 1.  If the refcount does
    // not drop to 1 when the function returns, but we did
    // set is_ref to 1 here, that means that is_ref must
    // already have been 1 to start with (since if it had
    // not, that means that the variable would have been
    // in a copy-on-write set, and would have been
    // seperated above).
    (*p_arg)->is_ref = 1;

    args_ind[af_index] = p_arg;

    assert (!in_copy_on_write (*args_ind[af_index]));
    args[af_index] = *args_ind[af_index];
  }
  else
  {
    zval *arg;
    if (local_f == NULL)
      arg = EG (uninitialized_zval_ptr);
    else
      arg = local_f;

    args[af_index] = fetch_var_arg (arg, &destruct[af_index]);
    if (arg->is_ref)
    {
      // We dont separate since we don't own one of ARG's references.
      arg = zvp_clone_ex (arg);
      destruct[af_index] = 1;

      // It seems we get incorrect refcounts without this.
      // TODO This decreases the refcount to zero, which seems wrong,
      // but gives the right answer. We should look at how zend does
      // this.

      arg->refcount--;
    }
    args[af_index] = arg;

    args_ind[af_index] = &args[af_index];
  }
  af_index++;


  phc_setup_error (1, "listings_source.php", 8, NULL TSRMLS_CC);

  // save existing parameters, in case of recursion
  int param_count_save = printf_fci.param_count;
  zval ***params_save = printf_fci.params;
  zval **retval_save = printf_fci.retval_ptr_ptr;

  zval *rhs = NULL;

  // set up params
  printf_fci.params = args_ind;
  printf_fci.param_count = 1;
  printf_fci.retval_ptr_ptr = &rhs;

  // call the function
  int success = zend_call_function (&printf_fci, &printf_fcic TSRMLS_CC);
  assert (success == SUCCESS);

  // restore params
  printf_fci.params = params_save;
  printf_fci.param_count = param_count_save;
  printf_fci.retval_ptr_ptr = retval_save;

  // unset the errors
  phc_setup_error (0, NULL, 0, NULL TSRMLS_CC);

  int i;
  for (i = 0; i < 1; i++)
  {
    if (destruct[i])
    {
      assert (destruct[i]);
      zval_ptr_dtor (args_ind[i]);
    }
  }


  // When the Zend engine returns by reference, it allocates a zval into
  // retval_ptr_ptr. To return by reference, the callee writes into the
  // retval_ptr_ptr, freeing the allocated value as it does.  (Note, it may
  // not actually return anything). So the zval returned - whether we return
  // it, or it is the allocated zval - has a refcount of 1.

  // The caller is responsible for cleaning that up (note, this is unaffected
  // by whether it is added to some COW set).

  // For reasons unknown, the Zend API resets the refcount and is_ref fields
  // of the return value after the function returns (unless the callee is
  // interpreted). If the function is supposed to return by reference, this
  // loses the refcount. This only happens when non-interpreted code is
  // called. We work around it, when compiled code is called, by saving the
  // refcount into SAVED_REFCOUNT, in the return statement. The downside is
  // that we may create an error if our code is called by a callback, and
  // returns by reference, and the callback returns by reference. At least
  // this is an obscure case.
  if (signature->common.return_reference
      && signature->type != ZEND_USER_FUNCTION)
  {
    assert (rhs != EG (uninitialized_zval_ptr));
    rhs->is_ref = 1;
    if (saved_refcount != 0)
    {
      rhs->refcount = saved_refcount;
    }
    rhs->refcount++;
  }
```

Outline

Original Speed-up

0.1x

(10 times slower than the PHP interpreter)

The problem with copies

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

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

Optimization

- Constant folding

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

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

Optimization

- Constant folding
- Constant pooling

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

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);
}
```

Optimization

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

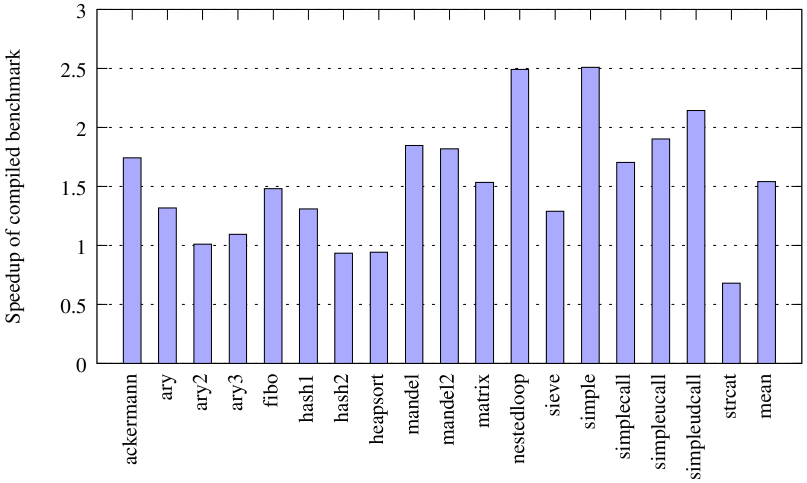
```
// $i = 0;
{
  zval* p_i;
  php_hash_find (LOCAL_ST, "i", 5863374, p_i);
  php_destruct (p_i);
  php_allocate (p_i);
  ZVAL_LONG (*p_i, 0);
}
```

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



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`