Things to learn from HPy

HPy Team
hpy-dev@python.org

Python Core Dev Sprint – C API Summit
October 9, 2023

Motivation for this Talk

enable more **performance optimizations** in CPython

<u>and at the same time</u>

provide a C **extension** API that can compile to

- a stable ABI
- an implementation-specific fast ABI

Concepts used in HPy that can help CPython

(i.e. if you ever do break the ABI, these are the things we'd like you to consider doing)

1. [API] Opaque handles
2. [API] Local vs non-local handles
3. [ABI] Explicit context argument with function table

Opaque Handles

- There **can** be several distinct handles denoting the same object

```
HPy y = HPy_Dup(x);     // may return a different handle
x == y;                 // compiler error
HPy_Close(y);           // matches handle that was dup’d (scoped)
```

Opaque Handles

- There **can** be several distinct handles denoting the same object

```
HPy y = HPy_Dup(x);
HPy_Is(x, y);
HPy_Close(y);
```

Opaque Handles

- There **can** be several distinct handles denoting the same object

```
HPy y = HPy_Dup(x);
HPy_Is(x, y);
HPy_Close(y);
```

- HPy’s CPython implementation just stores `PyObject*` in it

```
typedef struct _HPy_s { PyObject* _i; } HPy
```

Opaque Handles

- There **can** be several distinct handles denoting the same object

```
HPy y = HPy_Dup(x);          PyObject *y = x; Py_INCREF(y);
HPy_Is(x, y);                x == y;
HPy_Close(y);                Py_DECREF(y);
```

Implementation specific ABI

- HPy's CPython implementation just stores `PyObject*` in it

```
typedef struct _HPy_s { PyObject* _i; } HPy
```

Opaque Handles

- There **can** be several distinct handles denoting the same object

```c
HPy y = HPy_Dup(x);
HPy_Is(x, y);
HPy_Close(y);
```

```c
PyObject *y = x; Py_INCREF(y);
x == y;
Py_DECREF(y);
```

Implementation specific ABI

- HPy's CPython implementation just stores `PyObject*` in it

```c
typedef struct _HPy_s { PyObject* _i; } HPy
```

- BUT we can now experiment with other GC strategies (e.g. WASM, moving)
  - With indirection GC can move the object
  - This indirection is why handles are sometimes considered slower [1]

[1] Nanjekye J., et al., *Towards Reliable Memory Management for Python Native Extensions* (ICOOOLPS 2023).

Opaque Handles ⇒ tagged pointers

```
HPy HPy_AddImpl(HPy a, HPy b) {
  if (isTaggedInt(a) && isTaggedInt(b)) {
    return tagInt(untagInt(a) + untagInt(b));
  } else {
    return py2h(PyNumber_Add(a._i, b._i));
  }
}
```

- Can also do NaN boxing, list storage strategies, etc [https://doi.org/10.1145/2544173.2509531]

Local vs Non-local Handles

```
void * HPy_AsStruct(HPy x) { return (void *)x._i; }

void setName(HPy hpt, HPy name) {
  MyPerson *pt = (MyPerson *)HPy_AsStruct(hpt);
  pt->name = HPy_Dup(name); // BAD! Handles should be short-lived
}
```

Local vs Non-local Handles

```
void * HPy_AsStruct(HPy x) { return (void *)x._i; }

void setName(HPy hpt, HPy name) {
  MyPerson *pt = (MyPerson *)HPy_AsStruct(hpt);
  pt->name = HPy_Dup(name); // BAD! Handles should be short-lived
}
```

- Local handles are scoped
  - Only valid in the context of the current Python->C call
  - Implies: thread local
  - Arena (de-)allocations: Fast, good for NOGIL
- Non-local handles are explicit
  - Non-local handles are known, runtime can trace them w/o tp_traverse
    => e.g. some GC (Java, WASM) cannot call into tp_traverse

Local vs Non-local Handles

```
void * HPy_AsStruct(HPy x) { return (void *)x._i; }

void setName(HPy hpt, HPy name) {
  MyPerson *pt = (MyPerson *)HPy_AsStruct(hpt);
  HPyField_Store(hpt /*owner*/,
      &pt->name /*location*/, name /*handle*/);
}
```

- Local handles are scoped
  - Only valid in the context of the current Python->C call
  - Implies: thread local
  - Arena (de-)allocations: Fast, good for NOGIL
- Non-local handles are explicit
  - Non-local handles are known, runtime can trace them w/o tp_traverse
    => e.g. some GC (Java, WASM) cannot call into tp_traverse

Local vs Non-local Handles

```c
void * HPy_AsStruct(HPy x) { return (void *)x._i; }

void setName(HPy hpt, HPy name) {
  MyPerson *pt = (MyPerson *)HPy_AsStruct(hpt);
  HPyField_Store(hpt /*owner*/,
      &pt->name /*location*/, name /*handle*/);
}
```

```c
MyPerson *pt = (MyPerson *)hpt;
PyObject *tmp = pt->name; Py_INCREF(name);
pt->name = name; Py_XDECREF(pt->name);
```

Implementation specific ABI

- Local handles are scoped
  - Only valid in the context of the current Python->C call
  - Implies: thread local
  - Arena (de-)allocations: Fast, good for NOGIL
- Non-local handles are explicit
  - Non-local handles are known, runtime can trace them w/o tp_traverse
    => e.g. some GC (Java, WASM) cannot call into tp_traverse

Explicit Context Argument

- It can carry call-specific information
  - Including interpreter state
  - Provide handles for built-in objects like `None`

What Petr said…

Let’s talk about ABI stability

Explicit Context Argument **with Function Table**

- Why Function table? More flexibility than native linker
  - Specialized code: debug mode, tracing, … (instead of if-else-if cascade in the entry-points)
  - Runtime generated code: inline caches, dynamic tracing, …
  - Embedded systems without linker or with non-standard linker
- Why in the context?
  - Specialized code per call-site
    - Quickened call bytecode e.g. passes context with specializing **`HPy_CallMethod`** with inline cache to native extension call target => “JIT into C extensions”
  - Multiple incompatible API versions in one process
  - Easier to forbid calling (some) API functions
    - Forbid some API calls e.g. in tp_traverse, any call in random C threads (...)
    - Debug mode changes the pointers on purpose to detect misbehaving code

ABI stability **with Function Table**

```
HPy y = HPy_Dup(x);
HPy_Is(x, y);
HPy_Close(y);
```

```
typedef struct _HPy_s { PyObject* _i; } HPy
PyObject *y = x; Py_INCREF(y);
x == y;
Py_DECREF(y);
```

Implementation specific ABI

```
typedef struct _HPy_s { intptr_t _i; } HPy
HPy y = ctx->Dup(ctx, x);
ctx->Is(ctx, x, y);
ctx->Close(ctx, y);
```

Universal (stable) ABI

Debug mode **with Function Table**

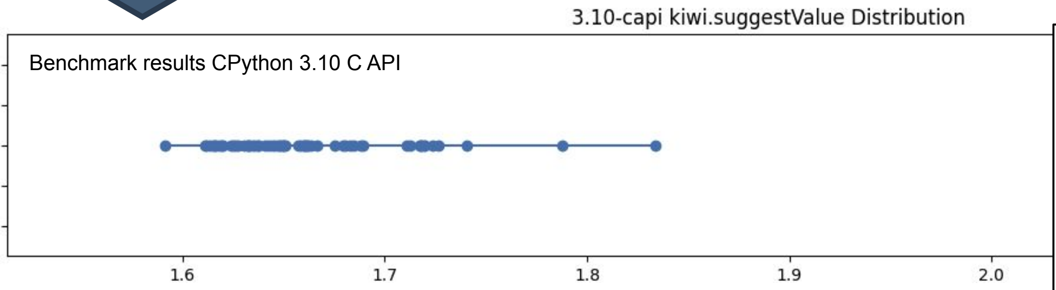
```
$ HPY=debug pytest -s -k test_constraint_creation py/tests/test_constraint.py
```

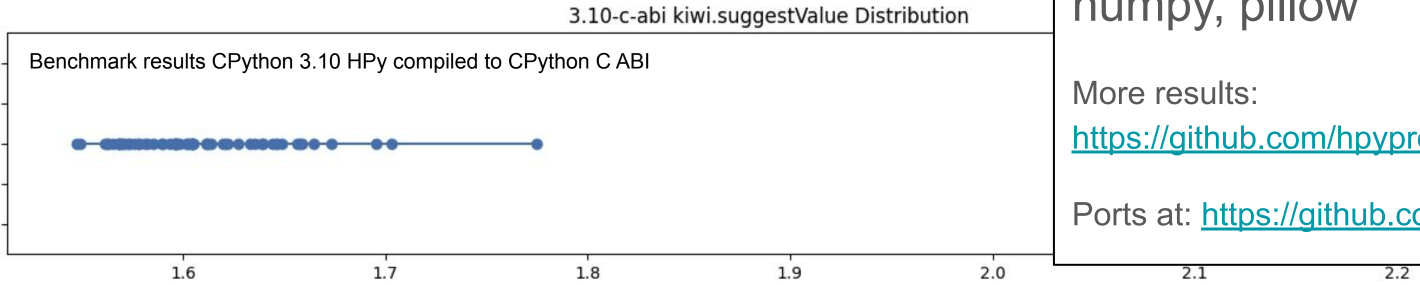
```
template<> inline
HPy BinaryAdd::operator()( HPyContext *ctx, Variable* first, double second, HPy h_first, HPy h_second )
{
  HPy temp = BinaryMul()( ctx, first, 1.0, h_first, HPy_NULL );
  if( HPy_IsNull(temp) )
      return HPy_NULL;
  return operator()( ctx, Term_AsStruct( ctx, temp ), second, temp, h_second );
}
```

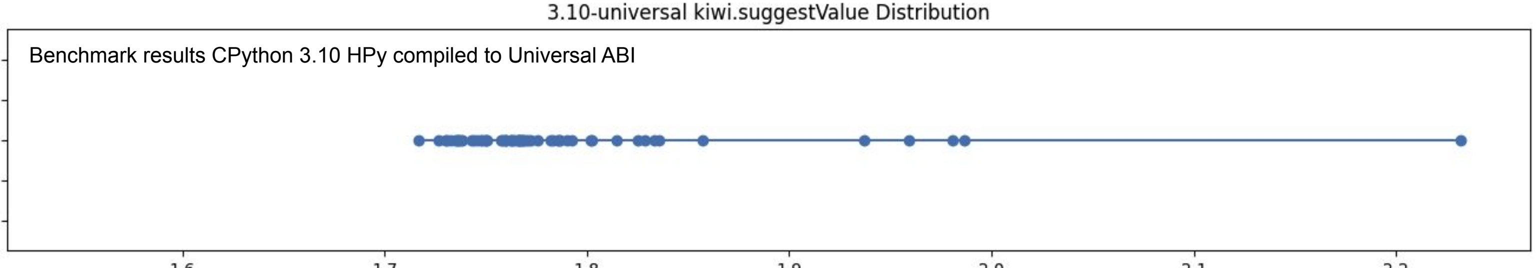
```
>  raise HPyLeakError(leaks)
E  hpy.debug.leakdetector.HPyLeakError: 10 unclosed handles:
E    <DebugHandle 0x5606bb9f81b0 for 1 * foo>
E  Allocation stacktrace:
E  python3.8/site-packages/hpy/universal.cpython-38d-x86_64-linux-gnu.so(debug_ctx_New 0x60) [0x7f4af12793c1]
E  kiwisolver.hpy.so( 0x67fc5) [0x7f4af0eaffc5]
E  kiwisolver.hpy.so(kiwisolver::new_from_global(_HPyContext_s*, HPyGlobal, void*) 0x55) [0x7f4af0eb236f]
E  kiwisolver.hpy.so(_HPy_s kiwisolver::BinaryMul::operator()<kiwisolver::Variable*, double>(_HPyContext_s*,
kiwisolver::Variable*, double, _HPy_s, _HPy_s) 0x4e) [0x7f4af0ebbc4e]
E  kiwisolver.hpy.so(_HPy_s kiwisolver::BinaryAdd::operator()<kiwisolver::Variable*, double>(_HPyContext_s*,
kiwisolver::Variable*, double, _HPy_s, _HPy_s) 0x5e) [0x7ff3e41cdba4]
```

Backup slides
(there are details and optimizations there)

Status of these ideas in HPy: We think it works







Complete ports: kiwisolver, ujson, piconumpy

Partial ports: matplotlib, cython, numpy, pillow

More results: https://github.com/hpyproject/hpy/wiki/dev-call-20220407

Ports at: https://github.com/orgs/hpyproject/repositories

Explicit Context Argument with Function Table

- The Debug Context

```
static inline HPy HPyLong_FromLong(HPyContext *ctx, long x) {
    return ctx->ctx_Long_FromLong(ctx, x);
}
```

1

2

```
DHPy debug_ctx_Long_FromLong(HPyContext *dctx, long value) {
     HPy uresult = HPyLong_FromLong(get_info(dctx)->uctx, value);
     return DHPy_open(dctx, uresult);
}
```

3

```
HPy ctx_Long_FromLong(HPyContext *ctx, long value) {
    return _py2h(PyLong_FromLong(value);
}
```

Explicit Context Argument with Function Table

- In HPy Universal ABI, the context is the function table **AND**
- It can carry call-specific information
  - Including interpreter state
  - Provide handles for built-in objects like `None`
- You can do decoration
  - HPy's debug/trace mode decorates functions to e.g. enforce contracts
- You can even do profiling
  - E.g. profile operands of binary operations (`HPy_Add`) or targets of calls (`HPy_CallMethod`)
  - Replace function pointer with a specialized function
  - Store specialized context near quickened call bytecode to HPy extension function

Explicit Context Argument

- Why do we need a context arg?
  - For ABI stability. For better performance. For better debugging.
- Can have ABI stability w/o a context?
- Basically yes, e.g. NumPy or SDL Library [1] do that
  -
  - They use a (hidden) global function table
  - An env variable allows to specify the ABI
- **BUT**
  - You anyway need a function table
  - For SDL, the table is global → one ABI per process
  - With a context, there can be an ABI per call
- Conclusion
  - Possible, way more complex
  - Passing `ctx` as first arg is easy and gives best performance

[1] https://www.reddit.com/r/linux_gaming/comments/1upn39/sdl2_adds_dynamic_api_magic_to_allow_updating_it/?rdt=61251

Concepts used in HPy that can help CPython

(i.e. if you ever do break the ABI, these are the things we'd like you to consider doing)

- Opaque handles
  - For Tagged and tagged values
  - For storage strategies
- Explicit context argument with function table
  - For explicit API contracts (debug context)
  - For profiling and specialization
- Local vs global handles that are not pointers to mutable objects
  - For alternative GC strategies (moving GC, WASM, request/response, arena collection)

How do Handles Affect the API?

- Opaque
  - No direct memory access like `((PyObject *)obj)->ob_type`
- No identity
  - You can't compare handles
  - You can't use them as unique key for objects

```
HPy h0 = HPyUnicode_FromString(...);
HPy h1 = HPy_Dup(ctx, h0);
memcmp(h0, h1, sizeof(HPy)) != 0
hash(h0) != hash(h1)
```

- Short-lived (scope: call)
  - Storing them in global vars is (in general) incorrect
  - Therefore: `HPyGlobal, HPyField`

```
static HPyGlobal g0;

void foo(HPyContext *ctx) {
    HPyGlobal_Store(ctx, &g0, ctx->h_None);
}
```

Local vs Global Handles: Better for NOGIL

- Correct us if we're wrong but `_Py_INCREF`'s complexity exploded, right?
  - Was formerly very simple and fast
  - With NOGIL, it is complicated and potentially expensive
  - If objects are shared, there will be a call

```
static _Py_ALWAYS_INLINE void
_Py_INCREF(PyObject *op)
{
    uint32_t local = _Py_atomic_load_uint32_relaxed(&op->ob_ref_local);
    if (_Py_REF_IS_IMMORTAL(local)) {
        return;
    }

    if (_PY_LIKELY(_Py_ThreadLocal(op))) {
        local += (1 << _Py_REF_LOCAL_SHIFT);
        _Py_atomic_store_uint32_relaxed(&op->ob_ref_local, local);
    }
    else {
        _Py_IncRefShared(op);
    }
}
```

Local vs Global Handles: Better for NOGIL

- In HPy, handles are scoped
  - Only valid in the context of the current call
  - Implies: thread local (and even stricter)
- Possible approach for CPython
  - The object pointer is aligned (let’s assume to 8 bytes)
  - Handles are opaque → code does not directly dereference the pointer
  - The 3 bits can be used for an index into a local ref count table

Local vs Global Handles

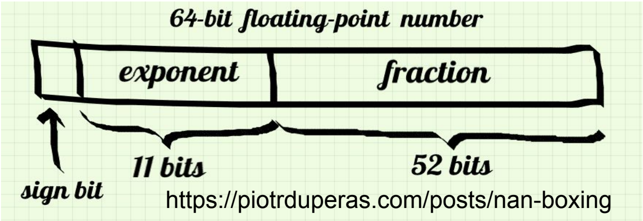
- You can use any lifetime management
- Reference counting
  - That's what we do in HPy's CPython impl
- Any tracing/moving/whatsoever garbage collector
  - That's what we do in PyPy and GraalPy
  - WASM GC?
- How can that work?
  - Well, the contract is more strict.
  - A Handle `h` denotes an object `o` but not vice versa
  - There can be several `h0, h1, h1` denoting the same object `o`
  - On CPython: HPy handle is a `PyObject *` but with more guarantees
- A handle does neither expose the object's location nor its identity
  - Move the object as you want or leave it there, it does not matter

Handles and NaN Boxing

- GraalPy implements that
- Handle is an index for a Java array → signed 32-bits
- Number conversion calls are for free (in many cases)
  - `HPyLong_(From|As)(U)Int(32|64)_t`
  - `HPyLong_(From|As)S(s)ize_t`
  - `HPyFloat_(From|As)Double`
- Getting the type of boxed values is super fast
  - There's room for a few interesting tags: tagged short strings, floats, ints, single element tuples
- Transparently extends the benefits of list storage strategies to C extensions (even on CPython) [https://doi.org/10.1145/2544173.2509531]

Opaque Handles ⇒ tagged ptrs, NaN boxing

- You can do NaN boxing, tagging, list storage strategies [https://doi.org/10.1145/2544173.2509531]
  - On 64-bit architecture, HPy has 64-bits width
  - IEEE 754
    - 64-bit floats are NaN if all of exponent bits are 1 and mantissa > 0, the remaining 51 bits don't matter




```
HPy HPy_Add(HPy a, HPy b) {
  if (isBoxedInt(a) && isBoxedInt(b)) {
    return boxInt(unboxInt(a) + unboxInt(b));
  } else {
    return PyNumber_Add(a._i, b._i);
  }
}
```

Performance vs ABI stability

More performant ⟷ More stable

- Intrinsic trade-off
- Different extensions/users, different needs
- Single API, multiple ABIs