Monte: Building a programming language using RPython

Corbin Simpson

October 10, 2015 (PyDX '15)

Hi!

I'm Corbin!

What you don't need to know:

- ✔ Monte
- ✔ RPython
- ✔ Lojban

You don't **need** to know compilers, but it will help.

Some Pre-talk Thanks

- ✔ Allen, for getting me to build Typhon
- ✔ Em and Mike, for trailblazing

The Python Side

- ✔ Twisted
- ✔ PyPy

Constant Sadness

- ✔ People don't use Twisted
- ✔ People don't use PyPy

Enter Monte

- ✔ Has syntactic and semantic features equivalent to always having Twisted available
- ✔ Reference implementation is built using PyPy's toolchain, **RPython**

RPython is not Python

What's RPython, Anyway?

- ✔ PyPy team: "RPython is a restricted subset of Python amenable to static analysis."
- ✔ Allen: "RPython is OCaml with a very odd syntax and a very odd standard library."
- ✔ RPython is a toolchain that can translate carefully crafted Python packages into highly efficient low-level code.

Why RPython?

You have to be writing an interpreter for a language; RPython is not easy to use.
If nothing else, know this: RPython turns interpreters into JIT compilers at a steep engineering discount.

Wait, What

Programs written in RPython are imported and then transformed from Python into a high-level statically-typed statically-named form ("translation"). This form is augmented:

- ✔ GC: A garbage collector, also written in RPython, is hooked in. The GC can be chosen at translation time.
- ✔ JIT: Based on JIT annotations written by hand, the program is turned into a JIT compiler which functions automatically and is correct independent of your program.

RPython finally performs some optimizations (malloc removal, generated switches, etc.) and emits an executable.

RPython Alternatives

GHC Haskell Speedy and terse but mutation is hard

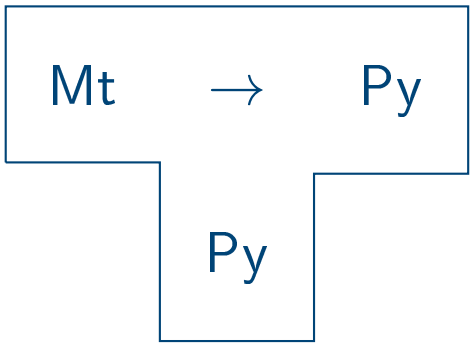
C++ Speedy but very difficult to write, read, and maintain

Truffle Very powerful but was not mature when I started

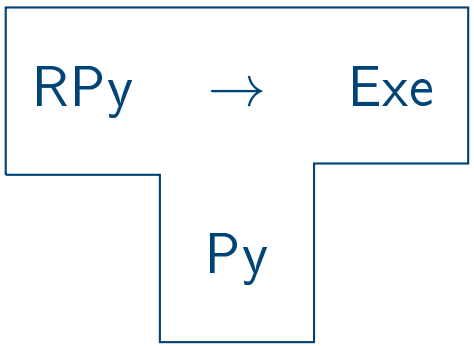
The Tombstones of Terror

Introducing Tombstones

The old Monte compiler:

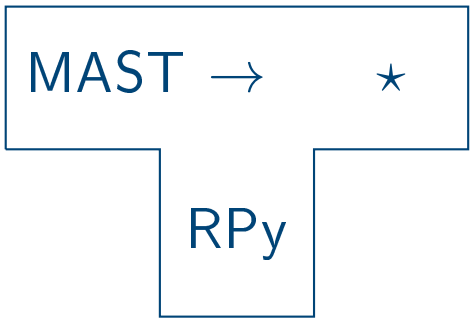


The RPython translator:

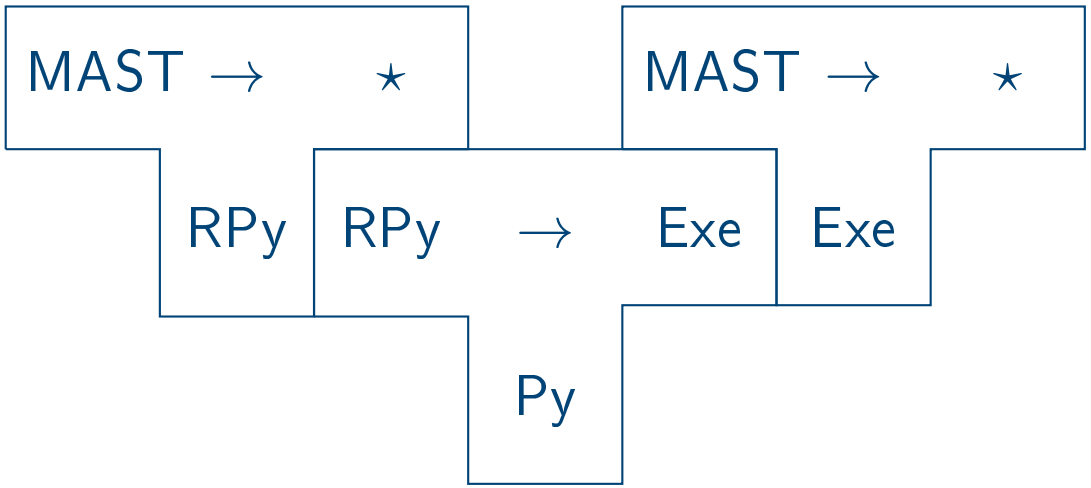


Untranslated & Translated

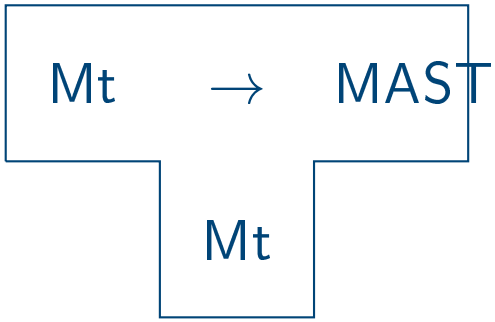
Typhon untranslated:



Typhon translated:



The Missing 'Stone



Typhon

It's Like an Onion

- ✔ AST loader
- ✔ AST-to-bytecode compiler
- ✔ Object model
- ✔ JIT annotations
- ✔ Vats and libuv

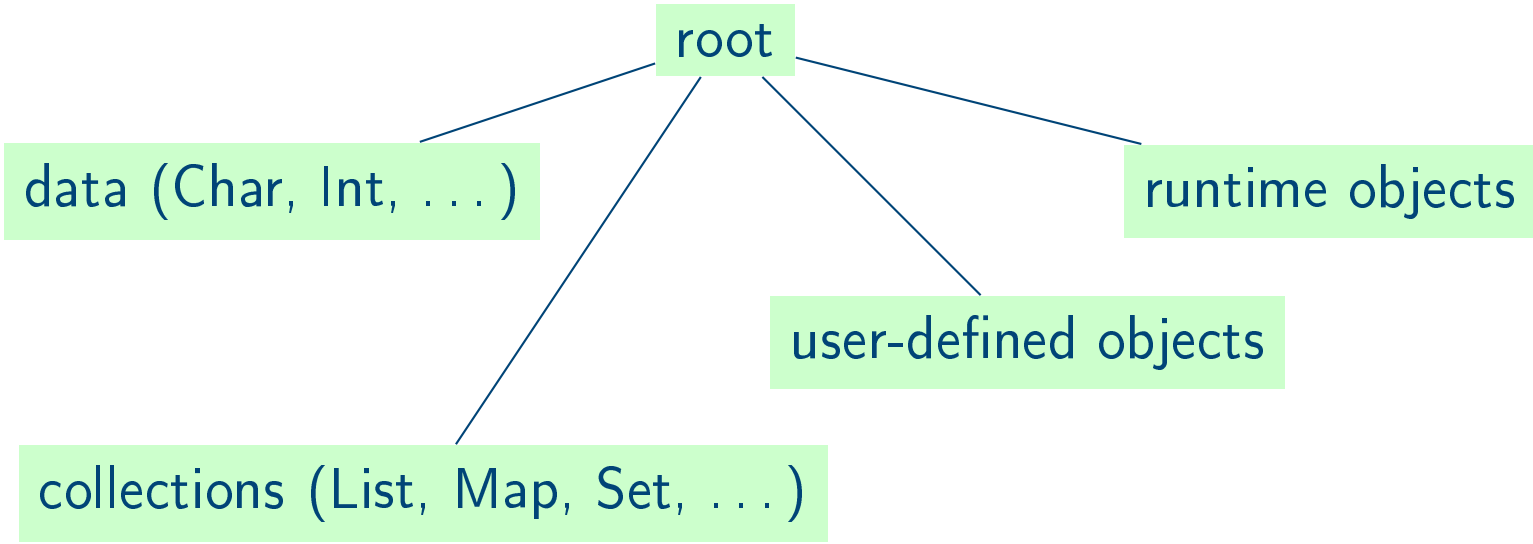
AST Loader

The AST loader loads files into an in-memory AST.

Bytecode Compiler

- ✔ Compiles to VM based on SmallCaps for E
- ✔ Compiler implemented as AST visitor
- ✔ Simple semantics: no loops, one test, exception mini-stack
- ✔ Compiler lowers static names to frame indices, turning dicts into lists
- ✔ Unstable design: Completely internal to Typhon (so it can be changed as needed)

Object Model



JIT Annotations

JIT's typical usage:

1. At every JIT merge point, check whether the current code is being executed often ("hot")
2. When code is hot, trace actions **of the interpreter** from one merge point to the next
   (a) When value or type discrimination occurs, **guard** the chosen branches
   (b) Trace through function calls (free inlining)
3. Optimize the trace, removing superfluous operations
4. When guards fail, retrace from the failed guard and create **branches**

JIT Annotations, Cont.

- ✔ The JIT needs to know which loops to trace. Place `merge_point` annotations at the head of each (user-level) loop.
- ✔ Any interpreter-level loop causes a function to be opaque to the JIT; it will be called and not traced. To fix this, use `unroll_safe`, but be wary of code explosion.
- ✔ Some functions need to be opaque for performance or to avoid JIT unsafety. They are marked `dont_look_inside`.
- ✔ Some functions need to be opaque to preserve referential transparency (while caching or otherwise being impure). They are marked `elidable`.
- ✔ "The trick": The JIT colors values as red or green depending on whether they are constant during the trace. `promote(red)` returns a green value; the JIT usually emits a guard.