PyPy's Approach to Virtual Machine Construction

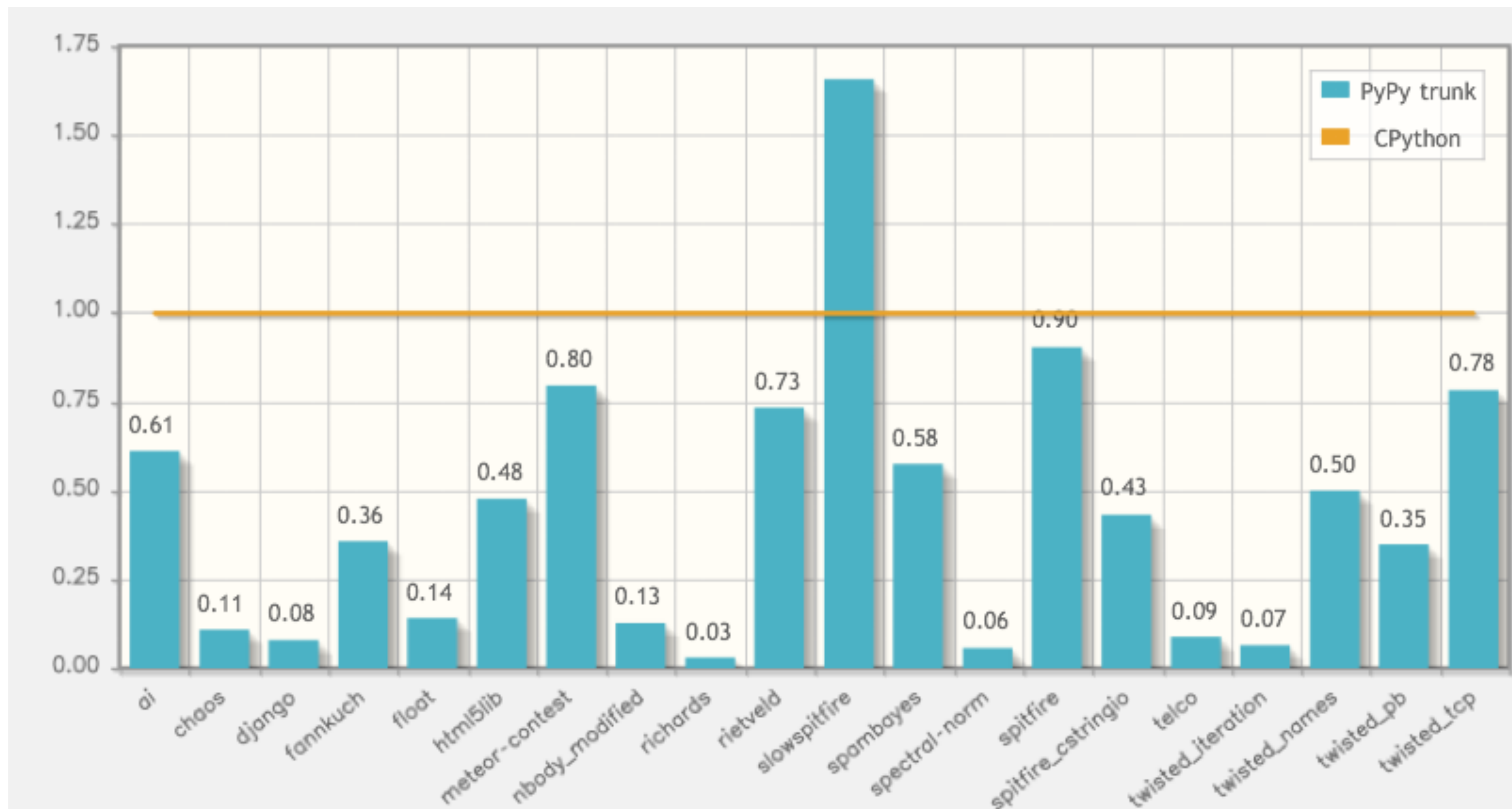
Armin Rigo, Samuele Pedroni

Prezentacja: Michał Bendowski

Czym jest PyPy?

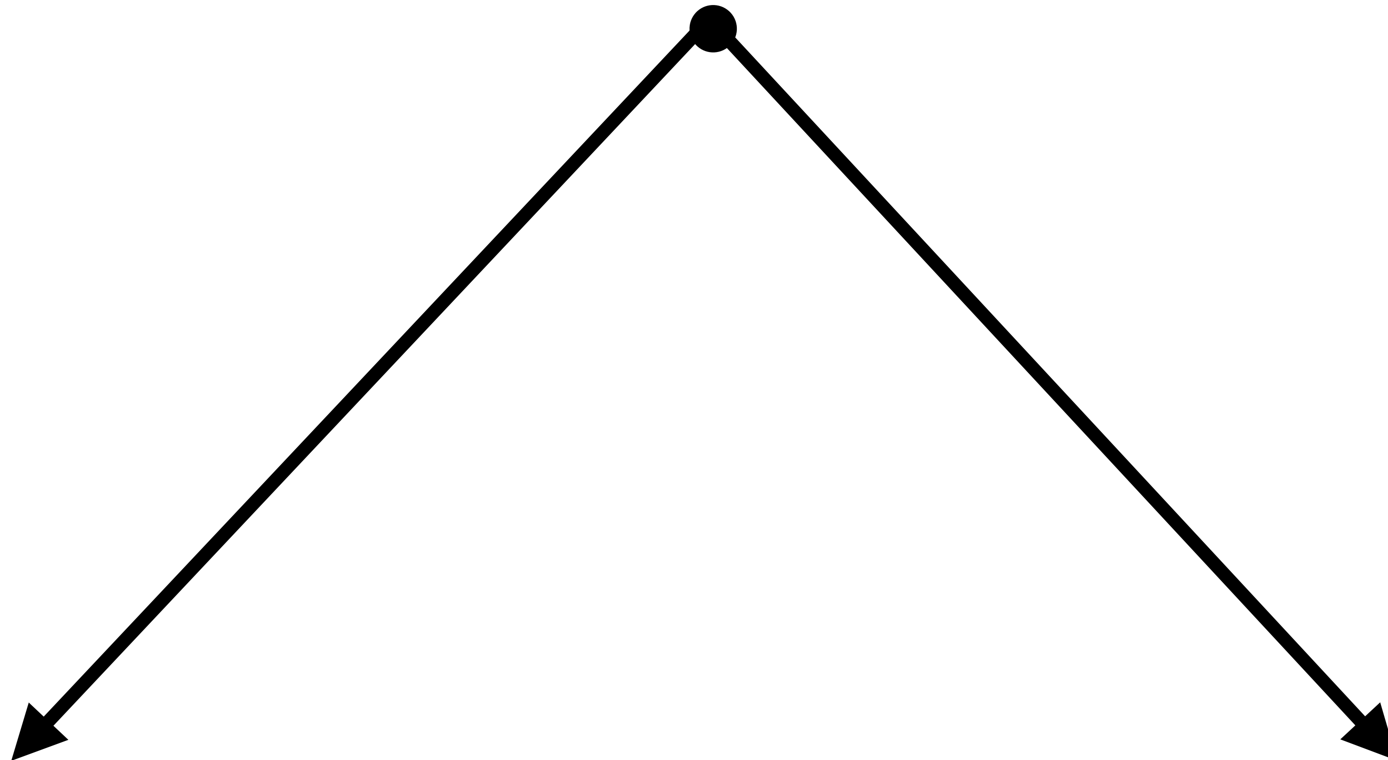
- Implementacja Pythona w Pythonie - wydajniejsza niż oryginalna implementacja w C
- Narzędzie pozwalające na pisanie wydajnych interpreterów w Pythonie - przeznaczonych zarówno dla x86 jak i JVM/CLR
- Udany projekt naukowy, prowadzony przez informatyków z wielu uniwersytetów

http://speed.pypy.org



Smaller is better

Z lotu ptaka



Interpreter Pythona napisany w (R)Python

Zestaw narzędzi tłumaczących dla języka RPython

RPython

- RPython = Restricted Python
- Podzbiór Pythona - każdy program w RPython można wykonać dowolnym interpreterem Pythona
- Kompromis między siłą wyrazu a potrzebą wydajności

RPython

- Wyjątki
- Pojedyncze dziedziczenie + domieszki (ang. mix-ins)
- Dynamiczne wyszukiwanie metod (ang. dispatch)
- Funkcje i klasy jako wartości pierwszej kategorii
- Standardowe struktury danych Pythona
- Brak refleksji w czasie wykonania (ale jest funkcja `isinstance`)
- Wiązania wewnątrz klas i globalnych przestrzeni nazw są stałe

RPython

- Programy w RPython są tłumaczone na kod w C lub bajtkod JVM, CLI/.NET
- Głównym zastosowaniem tego języka i towarzyszących mu narzędzi jest napisany w nim interpreter Pythona
- W ramach projektu PyPy powstały już implementacje Smalltalka, Scheme'a, JavaScriptu i innych języków

Kompilacja RPython

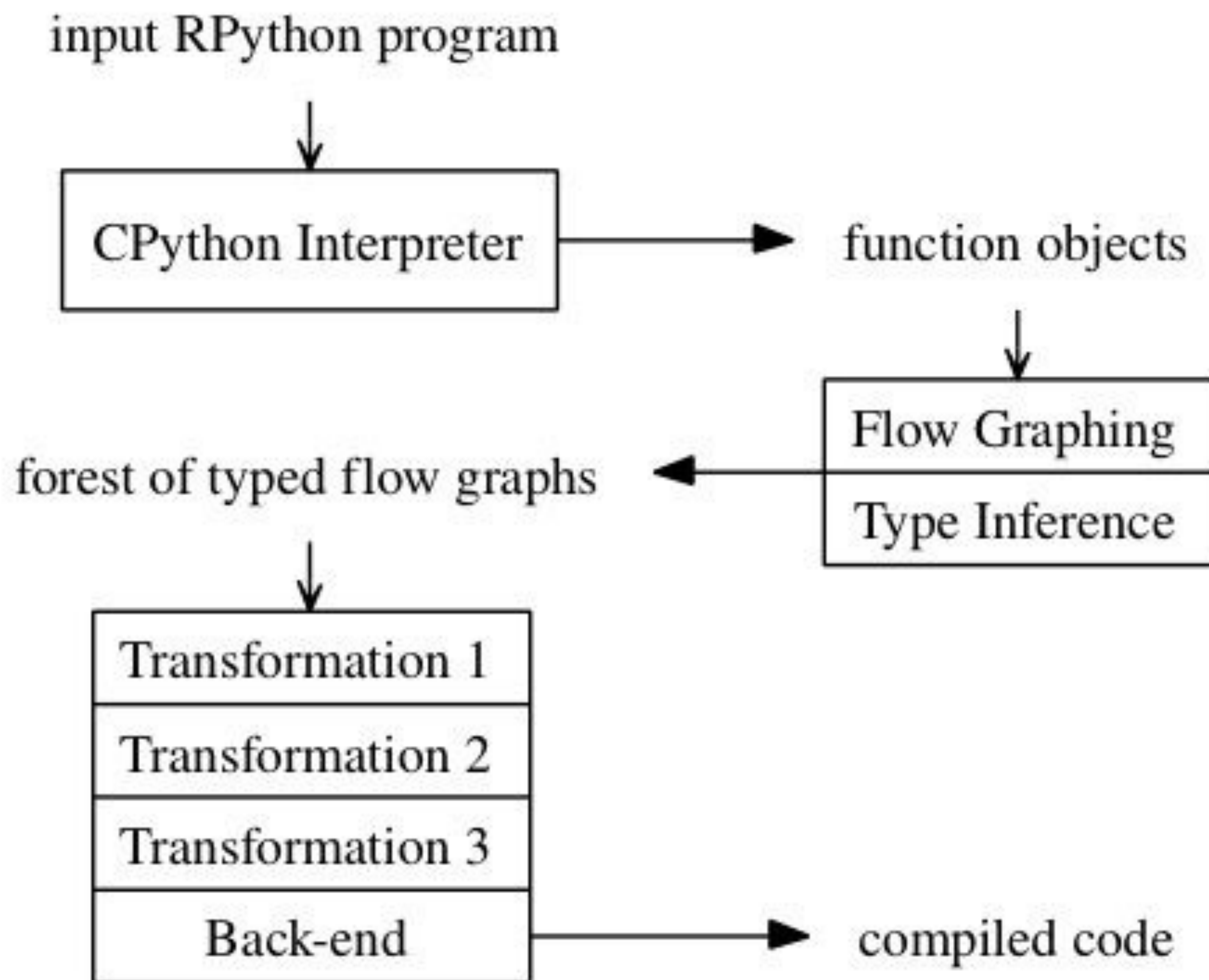
- Front-end produkuje grafy przepływu sterowania
- Do grafów dodawane są informaje o typach
- Seria transformacji operuje na grafach, czyniąc je coraz bardziej niskopoziomowymi
- Na koniec back-end generuje kod docelowy

Front-end

- Tekst programu trafia do interpretera Pythona (jak CPython lub `pypy`), który parsuje wejście i tworzy "function objects" - efektywną reprezentację zawierającą bajtkod Pythona (ten sam co w plikach `*.pyc`)

- Grafy przepływu sterowania budowane są przez wykonanie bajtkodu na specjalnych, "symulowanych" obiektach i zmuszenie interpretera, żeby zawsze wykonywał obie gałęzie instrukcji warunkowej

- Ten sam interpreter bajtkodu jest używany w "prawdziwym" interpreterze `pypy`


input RPython program
CPython Interpreter
function objects
Flow Graphing
Type Inference
forest of typed flow graphs
Transformation 1
Transformation 2
Transformation 3
Back-end
compiled code

- Pierwszym krokiem jest dopisanie informacji o typach. Na tym etapie system typów jest dość bogaty i zbliżony do kompletnego systemu typów Pythona

- Z czasem system typów będzie się zmieniał na bardziej restrtykcyjny, a grafy będą odpowiednio modyfikowane, żeby być z nim zgodne

Transformacje

- Pierwszą transformacją jest przejście na system typów docelowej platformy
- LLTyper - w przypadku C, obiektowość tłumaczymy na operacje na strukturach, z jawnym wskaźnikiem do tablicy metod wirtualnych
- OOTyper - w przypadku JVM obiektowość jest wbudowana w platformę, ale:
- oba powyższe współdzielą kod obsługujący semantykę Pythona i konwencje wołania metod i funkcji

Przykład transformacji - dodanie GC

- Można do tego podejść na kilka sposobów i wszystkie zostały zaimplementowane jako transformacje
- Zaczniemy od najprostszych
- W przypadku JVM/CLI ta transformacja oczywiście w ogóle nie jest potrzebna

Boehm GC

Możemy użyć gotowego Garbage Collector dla C i C++, takiego jak Boehm GC.

Wydajność będzie raczej słaba, ale ilość naszej pracy znikoma

Reference counting

Do grafów sterowania dopisujemy instrukcje zliczające referencje. Musimy też nieco zmodyfikować struktury danych.

To podejście nie jest bardzo wydajne, ale jest używane przez CPython i tam się sprawdza

Własny Garbage Collector

- Musimy zastąpić wywołania `malloc` odpowiednim mechanizmem z naszego GC
- Sam GC jest napisany w RPython
- Daje nam najwięcej okazji do optymalizacji pod kątem naszego zastosowania
- Obecnie stosowany w `pypy`

Coraz niżej

- Kolejne transformacje implementują operacje, które były uznawane za wbudowane na wyższych poziomach

- Na przykład: `lst.append(item)` jest tłumaczone na `ll_append(lst, item)`. Ta druga funkcja (napisana w RPython) operuje na obiektach symulujących tablice i wskaźniki, dzięki czemu może być wprost przetłumaczona na C

JIT compiler

- Jedną z najciekawszych transformacji jest możliwość dodania funkcji tracing JIT compiler do kompilowanego interpretera
- Programista musi jedynie wskazać, które zmienne należą do interpretera, a które do interpretowanego programu
- JIT obsługuje różne backendy, więc może być użyty również na CLR (wtedy emituje bajtkod CLR dla kluczowych śladów programu napisanego w Pythonie)

py.py

- Jako że cały kod jest napisany w (R)Pythonie, wszystkie komponenty systemu (interpreter Pythona, GC, biblioteka standardowa, kompilator RPythona, generator JIT) mogą być testowane i debugowane z użyciem istniejącej implementacji (CPython)
- Mogą być też rozwijane przez osoby najbardziej zainteresowane, czyli użytkowników języka Python

Dziękuję za uwagę