pythran : $\text{Python}_{2.7} \rightarrow \text{C++}_{11}$

Serge « sans paille » Guelton

QuietOceans / Télécom Bretagne / Namek

18 juin 2012

Avertissement

Les transparents projetés lors de cette séance pourraient heurter la sensibilité des puristes.

Les attraits du Python

- langage de haut niveau → abstraction
- langage interprété → bon $\frac{\text{SLOC}}{\text{heure}}$
- écosystème riche → favorise la réutilisation
- typage dynamique → concision, évite la redondance
- support natif des conteneurs classiques : `list`, `set`, `dict`

Parfait pour le prototypage rapide d'algorithmes

Mais

- code généré très lent (tout est dynamique)
- support du parallélisme limité (merci le GIL)

Approches

Pour avoir du code rapide

Réfléchir à ses algos

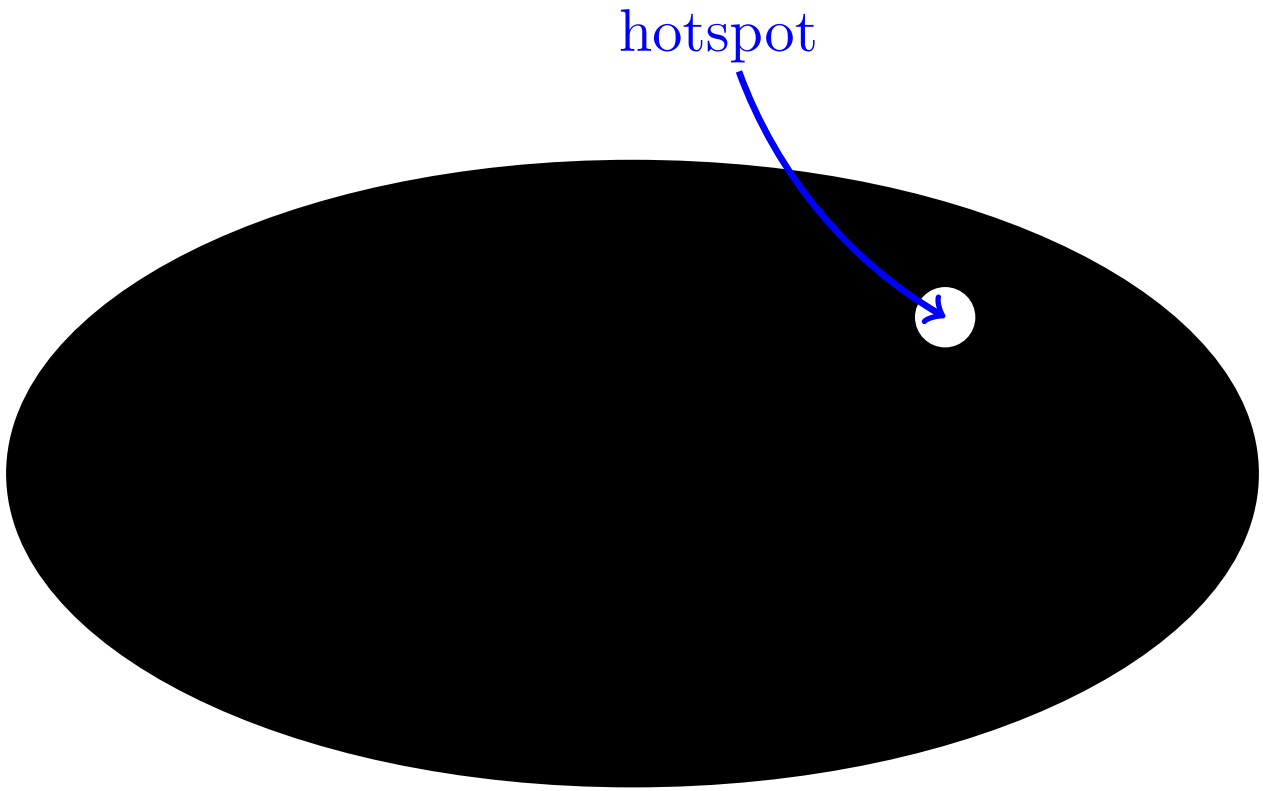
Il est recommandé d'utiliser des modules natifs

Pour avoir moult *threads*

Il faut passer en espace « natif »

La Pythie recommande de coder les *hotspots* en C/C++ et d'utiliser l'API native pour faire la passerelle avec le reste de l'application.

Vision (maintenant) classique d'une application



Problème : écrire tout le patatoïde en Python mais garantir de bonnes performances pour la partie critique.

Solutions existantes I

Shedskin : `http://code.google.com/p/shedskin`

- Convertit une application complète de Python vers C++
- Possibilité de générer des modules natifs
- Large support du langage

Mais pas d'élargissement de type, gestion mémoire faiblarde (`libgc`), pas de parallélisme

Copperhead : `http://copperhead.github.com`

- Convertit un DSL embarqué dans python vers du CUDA/OPENMP/TBB
- Génération à la volée de code natif *via* annotations

Mais langage d'entrée très restrictif et coût du JIT

Solutions existantes II

PyPy : http://pypy.org

- Interpréteur Python avec compilation JIT
- traduit à la volée du Rpython en C

Mais compilation dynamique uniquement, code généré séquentiel, artillerie lourde

---

https://github.com/serge-sans-paille/pythran

- Python sans classe vers C++
- Compilation statique
- Avec élargissement de type
- Parallélisant

Mais Surcoût de conversion de données entre Python et C++

Plan

Python, fonctions et C++

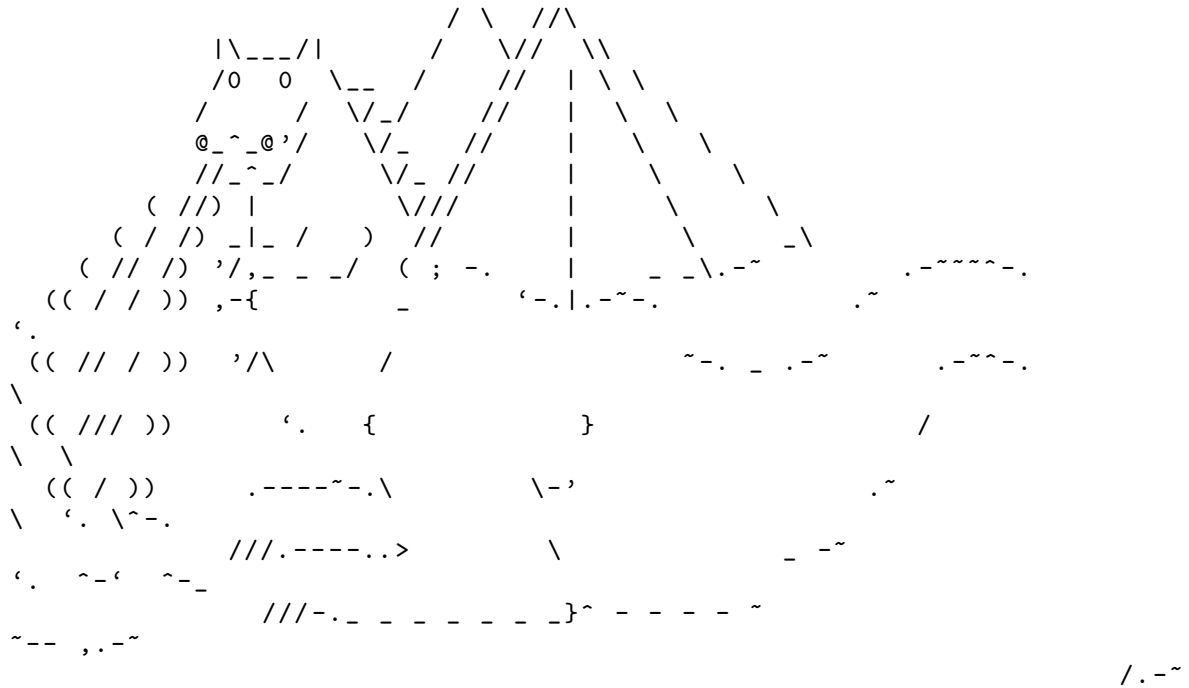
Typage en Python

typage structurel, dit « du canard »

Python, fonctions et C++

Typage en Python

typage structurel, dit « du canard »



Python, fonctions et C++

Typage en Python

typage structurel, dit « du canard »

Typage en C++

typage nominal, `Duck != Dragon`

mais : typage structurel *via* métaprogramation !

```
template<typename T>
void i_belive_i_can_fly(T& t) { t.fly(); }
...
Duck duck;
Dragon dragon;
i_belive_i_can_fly(duck);
i_belive_i_can_fly(dragon);
```

Fonctions polymorphes acceptées !

Exemple de fonction polymorphe en C++

```
template< class Op, class S0, class... Iters >
auto map_aux(Op& op, S0 const& seq, Iters... iters)
 -> sequence<decltype(op(*seq.begin(), *iters...))>
{
  decltype(_map(op,seq, iters...)) s;
  auto iter = std::back_inserter(s);
  for(auto & iseq : seq)
    *iter++= op(iseq, *iters++...);
  return s;
}

template <class Op, class S0, class... SN>
auto map(Op op, S0 const& seq, SN const &... seqs)
  -> decltype( _map(op, seq, seqs.begin()...) ) {
  return _map(op, seq, seqs.begin()...);
}
```

Pythran dispose d'un jeu de *headers*, `pythonic++`, fournissant la majorité des fonctions intrinsèques de Python sous forme de foncteurs.

Python, variables et C++

Variables en Python

variables polymorphes

```
a="bracadabra"
a=set()
a=1
```

Variables en C++

variables monomorphes

mais : extension implicite de type

Variables polymorphes refusées sauf si extension de type possible

Incursion dans le monde des variables polymorphes

`None`

En Python, toute variable peut être mise à `None`.
astuce : utilisation d'une construction similaire à `boost::optional`

Foncteurs

```
if c: a=lambda x,y:x+y
else: a=lambda x,y:x*y
```

Type nominal différent, type structurel identique :
→ possibilité de générer une interface commune !
mais implémentation similaire à `boost::variant` pour le moment.

Portées des variables

Variable Globale

bannies, utilisation disgracieuse en écriture en python, éventuellement à supporter en lecture seule ?

Fonctions imbriquées

convertie en fonction globales, avec capture du contexte

Portée des variables locales

En Python : la fonction mais le bloc en C++
$\rightarrow$ Déclaration en début de fonction...

Propagation de type

Pour chaque fonction

1. attribuer un type paramétrique par argument
2. propager ce type à travers l'AST
3. enregistrer les contraintes d'assignation
4. accumuler les contraintes

Au final chaque nœud de l'AST possède un type paramétrique.

Simplification et vérification de type

Délégué au système de typage du C++ !

- manipulation de types à l'aide de `decltype(`*`expr`*`)` et `std::declval<T>()`
- accumulation de contraintes à l'aide de `operator+`

Vérification de type

Le type de retour de la fonction suivante :

```
def dot_product(a,b):
  return sum(map(operator.mul, a, b))
```

S'exprime en C++ sous la forme

```
template<typename A, typename B>
struct $ {
  typedef decltype(
    pythonic::sum()(
      pythonic::map()(
        operator::multiply(),
        std::declval<A>(),
        std::declval<B>()
      )
    )
  ) return_type;
};
```

Contraintes interprocédurales

Les contraintes d'affectation ne sont pas toujours suffisantes.

```
-------------------------------------------
a=list()                  ||    a=list()
a+=[1]  # OK              //    a.append(1) # KO
-------------------------------------------
```

$\Rightarrow$ Modélisation des appels de méthodes comme des appels de fonction.

```
def append(a_list, a_value):...
```

$\Rightarrow$ Enregistrement des contraintes d'affectation sur des arguments et propagation interprocédurale.

Analyse de l'approche

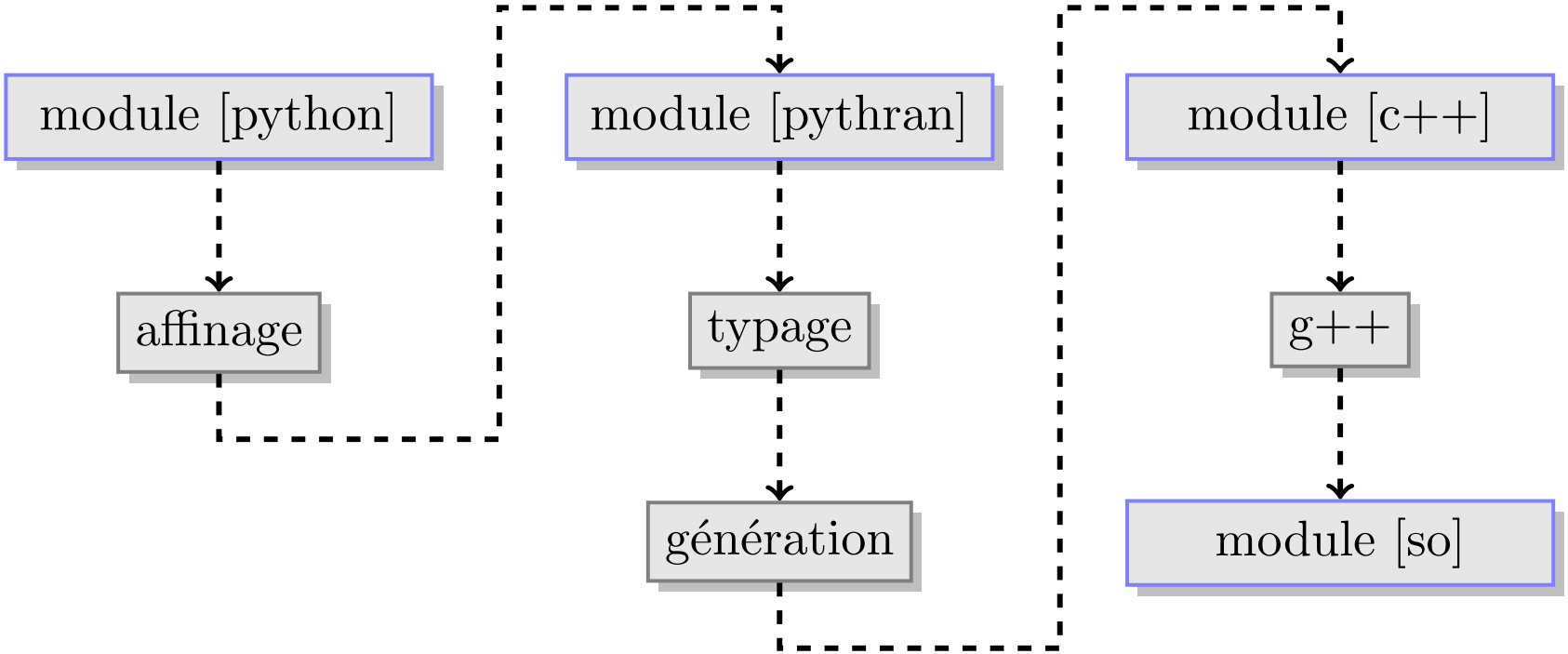
Réutilisation

L'inférence c++ est la référence, pas de savoir dans le moteur d'inférence.

Mais

- Gestion plus complexe de la récursivité (pas d'opérateur de fermeture)
- Pas d'étape de réduction / simplification
- Vérification de type difficile à tracer

Chaîne de compilation



affinage =

- nommage des fonctions anonymes
- suppression des fonctions imbriquées
- ...

Chaîne complète en python, pas en pythran...

Pythran dans une coquille de noix

pythran =	python
–	`class`
–	`eval`
–	introspection
–	variables polymorphes
–	modules [sauf `c?math`]

Support des exceptions, des ensembles et des dictionnaires en cours `[[ help accepted ]]`

Spécification de l'interface du module

module pythran = module python + spécification d'interface

Par exemple

```
#pythran export dot_product(int list, float list)
#pythran export dot_product(float list, float list)
def dot_product(a,b):
  return reduce(lambda x,y:x+y, [x*y for x,y in zip(a,b)])
```

Le mot clef `export` déclenche l'instanciation des fonctions *template* idoines.

Utiliser Pythran

Comme un préprocesseur

Pour inspecter le code C++ généré

```
$> pythran -E lent.py
```

Comme un compilateur

Pour générer un module natif

```
$> pythran lent.py -g -O2 -march=native -o rapide.so
```

Premières applications

Le bon

`hyantes` (code de géomatique) : code natif 8× plus rapide que le code d'origine, équivalent au code généré par `shedskin`

La brute

`matrix_multiply` : code natif 30× plus rapide que code d'origine, et 1.5× plus rapide que le code généré par `shedskin`

Le truand

`copy` : code 2× plus lent que le code d'origine.

Génération de code $\simeq$ 2× plus rapide que `shedskin` !

Vers la paralléisation

cette planche ne me coûte rien, rien n'est implémenté

`pythonic`

Certaines fonctions intrinsèques comme `sum` sont directement exprimable en OpenMP + avx

À grain moyen

En utilisant les structures du langage

(list comprehension|`map`|`reduce`)+fonction pure = boucles parallèles

À grain fin

Certaines fonctions polymorphes sont directement applicables sur des types vectoriels (cf. `boost::simd`)

Volet technique

Financements (non-ordonné)

HPC Project, QuietOceans, mon temps libre

Développement

Sources	`github.com/serge-sans-paille/pythran`
Canal IRC	FreeNode, `#pythran`
Sloccount	$\simeq 1K$SLOC [C++] $\simeq 3KSLOC$ [python] dont $1KSLOC$ de cas tests
Install	`python setup.py` ou *via* `easy_install`/Pypi

Conclusions et Perspectives

Conclusions

- Typage statique pour compilation statique
- *Backend* C++ pour la similitude des concepts

Perspectives

- Mise en œuvre des idées sur la parallélisation
- Diminuer le coût de conversion
- Autres modules à supporter ?
- Analyse de forme pour avoir des conteneurs plus efficaces

Les mots de la fin

Un grand merci...

- à la société SILKAN pour avoir financé une partie des travaux
- à mon employeur QUIETOCEANS pour m'avoir libéré du temps pour travailler sur Pythran
- aux organisateurs pour avoir financé ma participation
- à Fabien DAGNAT pour les nombreuses remarques constructives
- aux étudiants de Télécom Bretagne : Sébastien « Garrik » MARTINEZ, Pierrick BRUNET et Adrien MERLINI pour l'*alpha testing* et Adrien GUINET pour l'hébergement des premières heures