Optimisation optimiste à la volée pour les langages de programmation dynamiques

Maxime Chevalier-Boisvert

20 septembre, 2011

Université de Montréal

Langages de programmation

- Existent en plusieurs saveurs
- Haut vs bas niveau
- Typage statique vs dynamique
  - Annotations vs inférence de types
- Impératif, fonctionnel, logique
- Procédural, orienté-objet
- Usage général, spécifique au domaine
  - R, MATLAB

Langages de programmation dynamiques

- JavaScript, Python, Ruby, PHP, MATLAB, Perl, Scheme
- Sont récemment devenus très populaires (Google, etc.)
  - Dits plus faciles, plus expressifs, gains de productivité
- Pas aussi nouveaux qu'on pourrait penser
  - LISP, John McCarthy, 1958
- Caractéristiques importantes:
  - Typage dynamique
  - Résolution tardive
  - Génération de code à l'exécution (fonction *eval*)
  - Capacité d'introspection

Motivation

- Les langages dynamiques sont intéressants vu leurs avantages pratiques
- Performance inférieure aux langages statiques (C/C++, Java, etc.) dans presque tous les cas
  - Souvent par un ou deux ordres de magnitude
- Empêche l'utilisation des langages dynamiques dans les domaines où la performance est importante
- Beaucoup de sites web sont programmés en Python/Ruby/PHP du côté serveur
  - Réductions en matériel et en énergie possibles

Objectifs

- Tester une nouvelle approche pour l'optimisation des langages dynamiques
  - Optimisations optimistes
- Explorer plusieurs options d'implémentation et leur impact sur la performance
- Démontrer la viabilité de cette approche
- Rapprocher la performance des langages dynamiques de celle des langages statiques

Méthodologie

- Approche expérimentale
- Développement du compilateur Tachyon
- Intégration des optimisations optimistes à Tachyon
- Emphase sur JavaScript
  - Représentatif des langages dynamiques
  - Langage réel, mais relativement simple
- Étude empirique pour guider l'exploration

Critères de succès

- Validation de l'efficacité avec programmes de test
- Ensemble de programmes représentatifs
  - “There is no such thing as a representative JS benchmark”
  - Programmes “réels” préférés aux microbenchmarks
  - Plus de programmes, plus de couverture
- Objectifs
  - Amélioration sur tous les programmes par rapport au même compilateur sans optimisations optimistes
  - Gains majeurs sur certains programmes, se rapprochant de la compilation statique
  - Dépasser ou se rapprocher beaucoup des VMs commerciales

Tachyon

- Compilateur JavaScript métacirculaire
  - Écrit en JavaScript
  - JIT pur (pas d'interprète)
  - Compilation par fonction
  - Support pour x86 32/64-bit
- Projet initié au LTP en été 2010
- Plateforme de test pour nouvelles optimisations et idées de conception de langages
- Pour ma thèse, je développe Tachyon pour tester les *optimisations optimistes*

JavaScript: un court résumé

- JavaScript ≠ Java (pas de parenté)
- Syntaxe impérative (infixe)
- Orienté-objet
  - Héritage par prototypes (comme SELF)
- Composantes fonctionnelles
  - Fermetures, fonctions imbriquées, apply/map
- Langage dynamique
  - Typage dynamique, eval
  - Ajout/effacement de propriétés dynamiquement

Exemple de code JavaScript (1/3)

```
d8> var obj = { x:1, y:’foo’ }

d8> obj.z = 3
3

d8> for (k in obj) print(k);
x
y
z

d8> obj.hasProperty('w')
false

d8> delete obj.z
true

d8> print(obj.z)
undefined
```

Exemple de code JavaScript (2/3)

```
d8> function add(x) { return function (y) { return y+x; } }

d8> var arr1 = [11,22,33];

d8> var arr2 = ['book','bag','apple'];

d8> print(arr1.map(add(100)));
111,122,133

d8> print(arr2.map(add('s')));
books,bags,apples
```

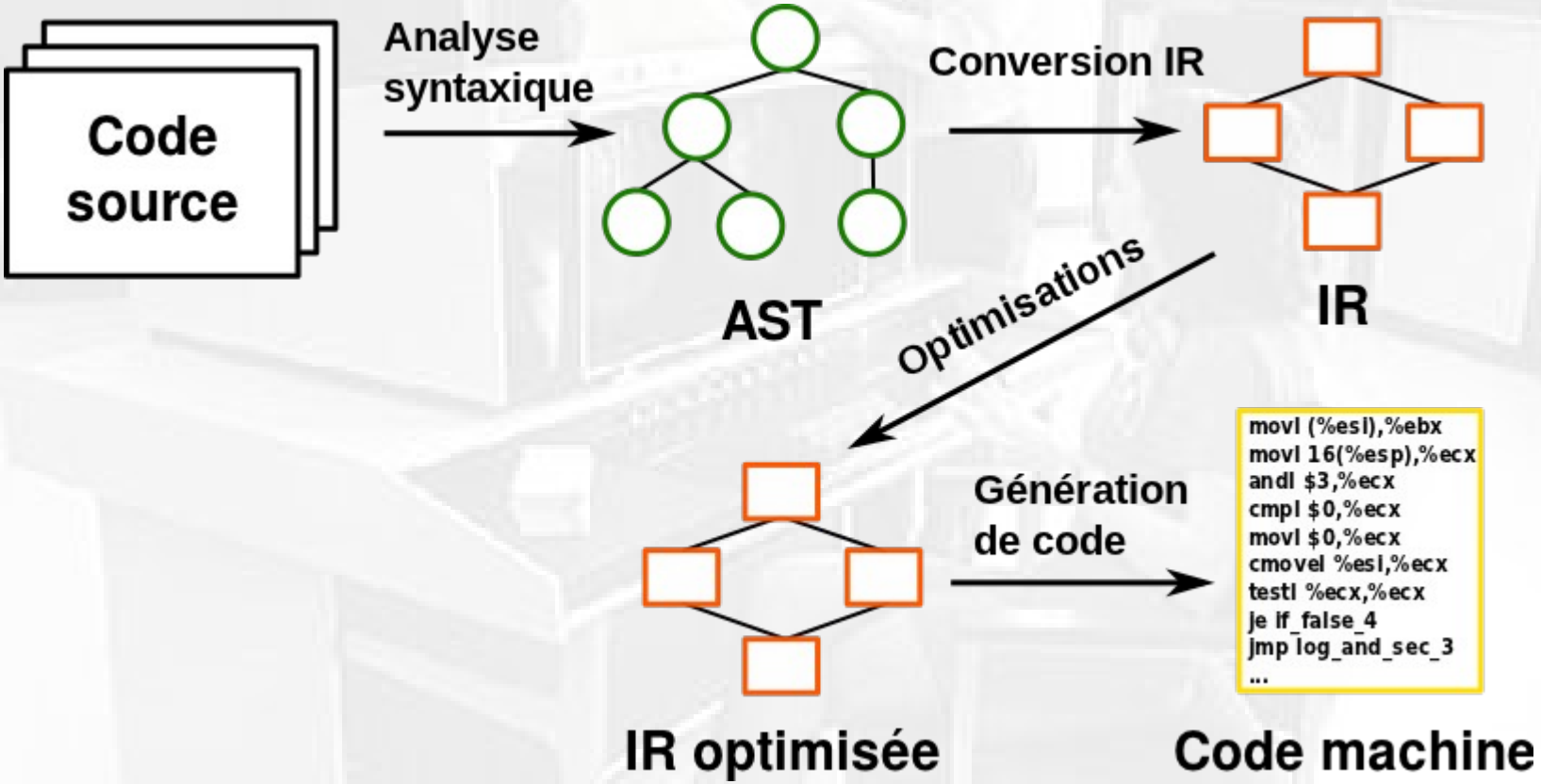
Exemple de code JavaScript (3/3)

```
d8> function foo() { print('Hi'!); }

d8> foo();
Hi!

d8> eval(read('foobar.js'));

d8> foo();
EXTERMINATE!
```

Optimisation de code

- Essentiel de l'optimisation en compilation:
  - Spécialiser le code
  - Éliminer les redondances
  - Implantation plus simple/rapide
- Traditionnellement: compilation statique, à l'avance
  - Analyser le programme à la compilation
  - Tester/prouver la validité d'un ensemble d'optimisations
  - Appliquer les optimisations valides

Modèle de compilation traditionnel



Validité des optimisations

- Pour appliquer une optimisation, on doit prouver qu'elle ne change pas la sémantique du programme pour tout état futur possible
- Doit être en mesure de déterminer ceci à l'avance, au moment de la compilation
- La seule source d'information est le code source
- Peut dériver certains faits avec des analyses
- Le typage statique aide beaucoup le compilateur
  - Restreint les comportements possibles du programme
  - Fourni davantage d'information pour l'optimisation

Optimisations traditionnelles et langages dynamiques

- Techniques traditionnelles mal adaptées
- Typage dynamique
  - Peu d'informations sur lesquelles baser nos analyses
  - Opérateurs, fonctions polymorphiques

    +, *, etc. s'appliquent à tous les types
- Chargement dynamique, eval
  - Rend toute analyse statique difficile
  - Nouveau code peut changer le programme
- Beaucoup de faits utiles sur le programme sont difficiles (voir impossibles) à prouver

Compilateurs à la volée (JIT)

- Retardent la compilation
  - Jusqu'à l'exécution du programme
  - Aussi longtemps que possible (compilation paresseuse)
- Génèrent du code machine pour l'architecture cible
- Performent facilement mieux qu'un interprète
- Avantage théorique sur les compilateurs statiques
  - Peuvent examiner l'état d'un programme à l'exécution
  - Opportunités inexploitées pour les langages dynamiques
- Désavantage: la compilation doit être rapide

Optimisation adaptative

- Idée: adapter les optimisations aux programmes
  - Traditionnellement, on applique toujours les mêmes optimisations de la même façon, en séquence (pipeline)
- Formule typique:

  compilation + profilage + compilation optimisée
- Spécialisation des types par profilage
  - Observer les types fréquents, spécialiser (SELF, V8)
- Analyse des caractéristiques d'un programme
  - Apprentissage machine (Agakov, iterative optimization)
- Réoptimisation sur demande
  - Plusieurs niveaux d'optimisation (HotSpot VM)

Compilateurs de traces

- Trace: groupe d'instructions fréquemment exécutées séquentiellement
- Extension d'un interprète avec un JIT
  - Compilation JIT des traces des boucles les plus exécutées
- Appliqués avec succès à l'optimisation des langages dynamiques
  - Mozilla TraceMonkey
  - LuaJIT
  - PyPy
- Avantages:
  - Linéarisation des chemins fréquents
  - Inlining automatique
  - Spécialization en fonction des types
  - Élimination de tests redondants

Limitations des compilateurs de traces

- Compilateurs de traces: orientés vers le code
  - Quelle séquences de code sont exécutées le plus fréquemment? Comment peut-on les rendre rapides?
  - Analyses et optimisations locales des traces, en isolation
- Pas de composante d'analyse globale. La spécialisation est purement locale
  - Beaucoup de tests redondants ne sont pas éliminés
- Je crois qu'une spécialisation “globale” est nécessaire pour maximiser la performance

Optimisations optimistes

- Langage dynamique $\rightarrow$ compilateur dynamique
- Traditionnellement, pour appliquer une optimisation:

  Prouver que l'optimisation ne change pas la sémantique du programme pour tout état futur possible
- Prouver la validité d'une optimisation à l'avance est difficile
  - Plus difficile pour les langages dynamiques
- Et si on se basait sur une propriété moins forte?
  - Optimisation est valide selon l'état actuel du programme
  - Pourrait ne pas être valide dans le futur

Réoptimisation à l'exécution

- Support pour la réoptimisation à l'exécution
- Enlève la difficulté de prouver la validité des optimisations de façon conservatrice
- Idée explorée dans d'autres contextes
  - Défaire les choix d'inlining dynamiquement
  - Optimistic interprocedural analysis framework (Sarkar, OOPSLA '01)
- Grand potentiel pour les langages dynamiques
  - Plus le dynamisme est élevé, plus on part de loin

Exemple d'optimisation (1/3)

```
var zero = 0;

function sum(list) {
    var total = zero;
    for (var i = 0; i < list.length; ++i)
    {
        var t = list[i];

        // Addition ou concaténation
        total = total + t;
    }
    return total;
}

function f(x) { zero = x; }

print(sum([1,2,3,4,5]));
```

Exemple d'optimisation (2/3)

```
var zero = 0;

function sum(list) {
    var total = zero;
    for (var i = 0; i < list.length; ++i) {
        var t = list[i];
        if (typeof total === 'number' && typeof t === 'number')
            total = numberAdd(total, t);
        else
            total = genericAdd(total, t);
    }
    return total;
}

function f(x) { zero = x; }

print(sum([1,2,3,4,5]));
```

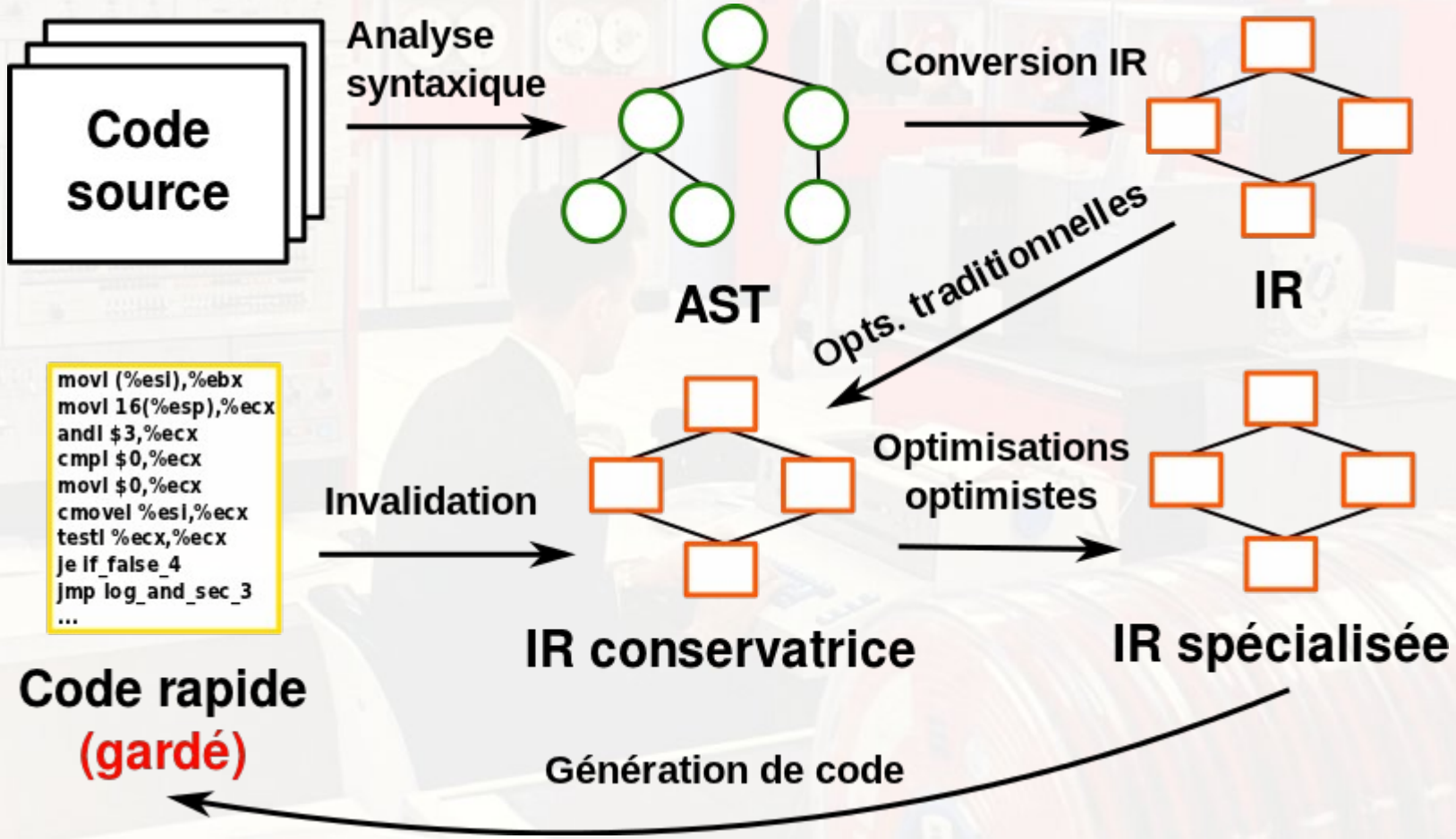
Exemple d'optimisation (3/3)

```
var zero = 0;

function sum(list) {
    var total = zero;
    for (var i = 0; i < list.length; ++i) {
        var t = list[i];
        total = numberAdd(total, t);
    }
    return total;
}

function f(x) {
    zero = x;
    if ((zero instanceof Number) === false)
        sum = eval(sum);
}

print(sum([1,2,3,4,5]));
```

Gardes et spéculation

- Peut obtenir de l'information de typage avec de l'inférence de types
  - Types inconnus: trous dans l'information fournie
- Peut combler ces trous avec information de profilage (types observés)
- Spéculation: types observés resteront valides
- Insertion de gardes pour détecter bris des suppositions optimistes
- Si suppositions invalidées, réoptimisation

Modèle de compilation optimiste



Propagation de types

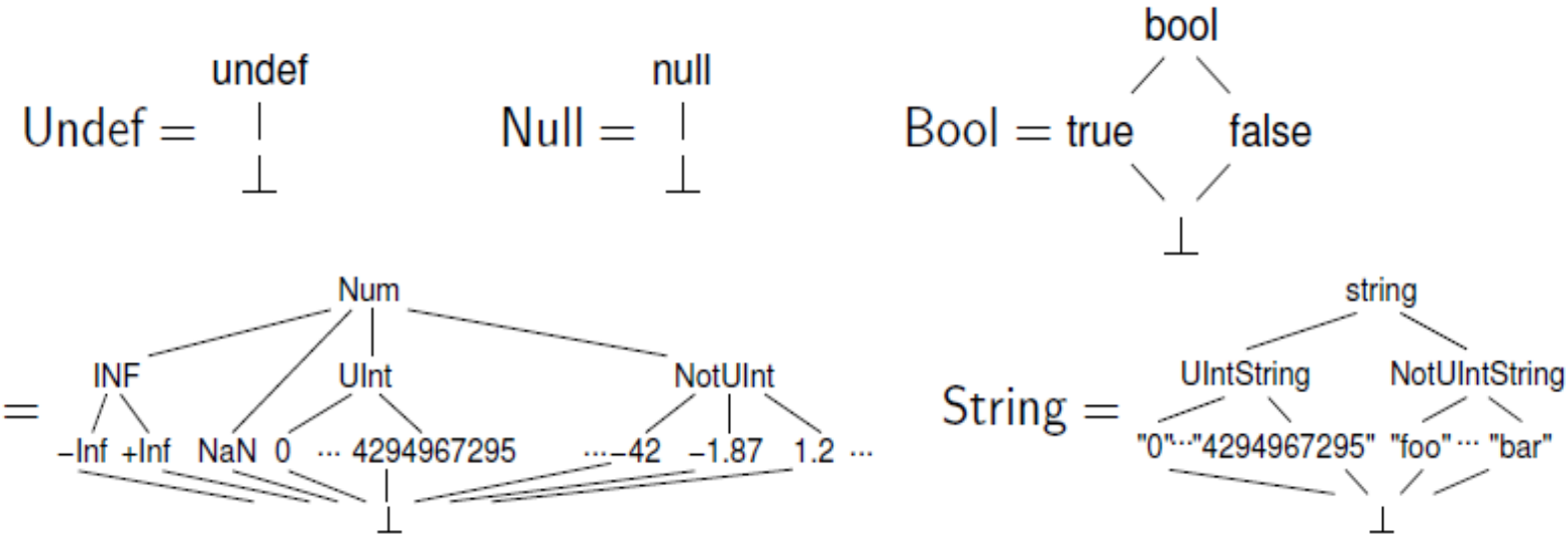
- Inférence de types sous forme d'analyse de flot
- Calcul de point fixe
- Analyse par fonction ou globale
- Hiérarchie de types
- Types de base JS:
  - string, entier, double, booléen, undefined, null, objet
- Types des objets:
  - Veut connaître les types possibles de chaque propriété
  - Division possible en classes d'équivalences

Caractérisation des types JavaScript (1/2)

Abstract values are described by the lattice Value:

$$\text{Value} = \text{Undef} \times \text{Null} \times \text{Bool} \times \text{Num} \times \text{String} \times \mathcal{P}(L)$$

The components of Value describe the different types of values.



For example, the abstract value $(\bot, \text{null}, \bot, \bot, \texttt{baz}, \emptyset)$ describes a concrete value that is either `null` or the string "`baz`", and $(\text{undef}, \bot, \bot, \bot, \bot, \{\ell_{42}, \ell_{87}\})$ describes a value that is `undefined` or an object originating from ℓ_{42} or ℓ_{87}.

De l'article *Type Analysis for JavaScript* (S. H. Jensen, A. Moller & P. Thiemann)

Caractérisation des types JavaScript (2/2)

- L'analyse de Jensen et. al est très lente et consomme énormément de mémoire
- Compromis entre vitesse et précision
- Plus rapide si on prend moins d'information en compte
  - Déterminer quelle information est la plus utile pour l'optimisation
- Plus rapide si on est moins sensible au contexte

La performance

- Potentiel d'améliorer la performance
  - Code optimisé beaucoup plus rapide
- Rajoutent un surcoût
  - Temps de profilage
  - Exécution des gardes
  - Temps de recompilation
- Quels programmes ont le plus de chance de bénéficier des optimisations optimistes?
  - Programmes qui s'exécutent longtemps

Recompilations en fonction du temps

- Quand une supposition optimiste est invalidée, il peut y avoir une recompilation
- Si on désoptimise le programme de façon monotone, une supposition ne peut être invalidée qu'une seule fois
- Plus un programme s'exécute longtemps, plus il explore les chemins d'exécution possibles
- Recompilation en fonction du temps
  - Courbe se rapprochant d'un asymptote
  - Multiples phases: courbe en escalier

“Type Inference” de Mozilla

- Brian Hackett de Mozilla travaille sur un projet similaire:

  “[...] whole-program, hybrid static and dynamic analysis that attempts to find the set of possible types [...]”

  “[...] If assumptions about type information are broken [this] can cause methods to be recompiled [...]”
- Résultats initiaux positifs sur tests V8 avec un prototype s'intégrant à JaegerMonkey
- Supporte la validité de cette idée

État actuel du projet

- Tachyon supporte la majorité d'ES5
  - Bibliothèque standard presque complète
  - Manque exceptions, nombres à virgule flottante, ramasse miettes, attributs de propriétés
- Tachyon peut se compiler lui-même
- Présenté à CASCON 2010
- Financement de Mozilla Corp.
- Publication acceptée à DLS 2011
- Prototype d'optimisation spéculative

Calendrier académique

- Conception de l'analyse de types (sept-déc 2011)
- Spécialisation automatique des primitives (jan-mar 2012)
- Système de profilage (été 2012)
- Recompilation de fonctions en exécution, optimisations optimistes (fin 2012)
- Développement et maintenance
  - Améliorations à l'IR
  - Compléter Tachyon (exceptions, FP, GC, etc.)
  - Ajouts optionnels à Tachyon (modules, etc.)

Perspectives à long terme

- Optimisations persistantes
  - L'optimisation implique un coût à l'exécution
  - Sauvegarde du programme optimisé
  - Exécutions futures commencent déjà optimisées
- Optimisation continue
  - Un programme peut changer son mode d'opération
    - i.e.: différentes phases, différentes données
  - Certains programmes exécutent pendant longtemps
    - e.g.: serveurs web
  - Réoptimisation quand nécessaire

En conclusion

- Je crois que les optimisations optimistes pourraient changer la façon dont les compilateurs JIT sont implantés
- Potentiel d'amener la performance des langages dynamiques beaucoup plus proche de celle des langages statiques
- Et si les langages dynamiques performaient aussi bien que les langages statiques?
  - Plus de compromis à ce niveau