AutoVM : Repousser les frontières de la généricité

Gaël Thomas – Maître de conférences

gael.thomas@lip6.fr

Université Pierre et Marie Curie

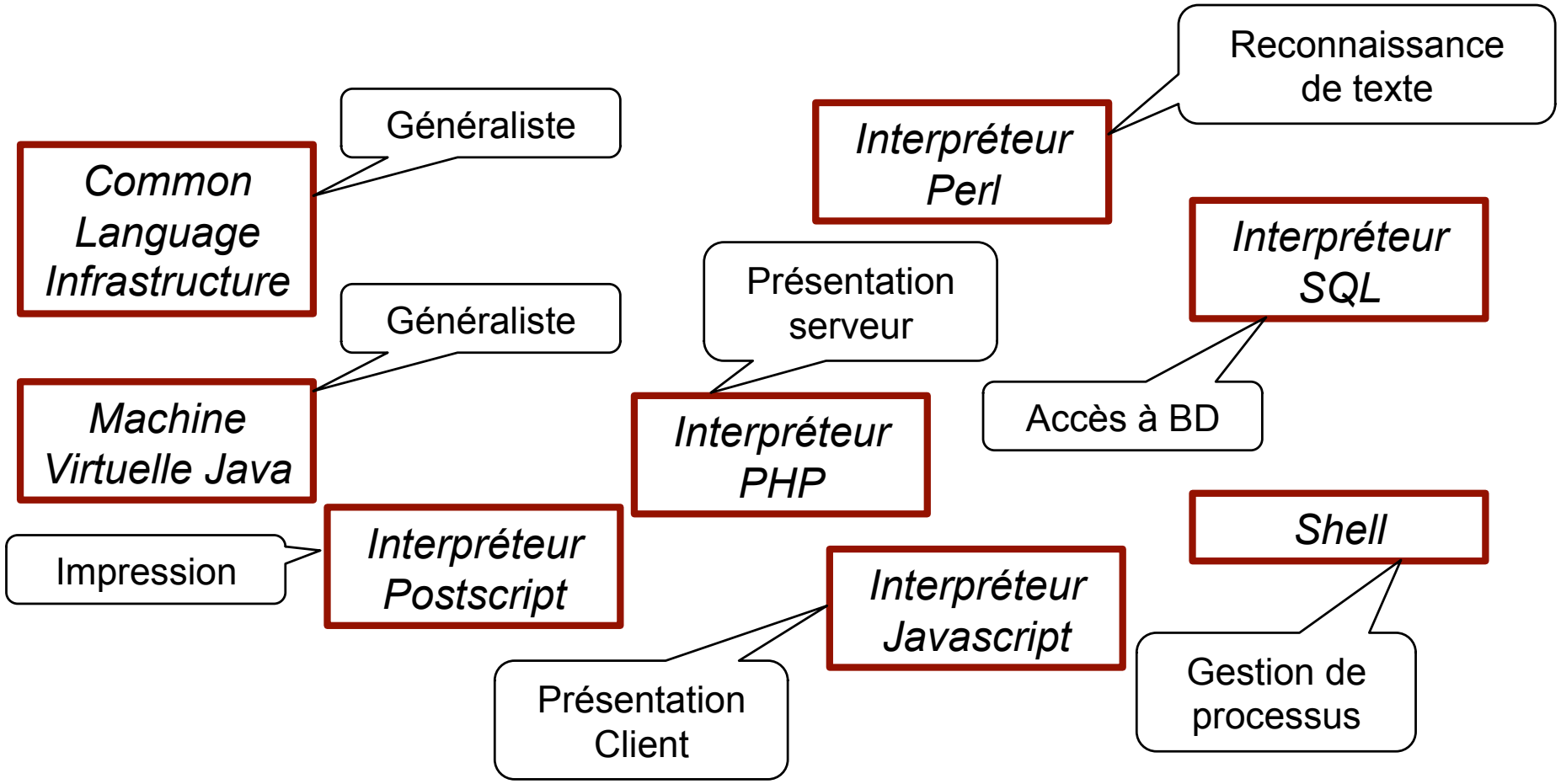
Equipe REGAL

LIP6/INRIA

Multiplication des VMs

Machine Virtuelle Applicative :

Environnement d'exécution pour code abstrait



Multiplication des VMs

Pourquoi un tel succès :

- *Exécution contrôlée par la machine virtuelle*
  - ✓ Vérification du typage
  - ✓ Vérification des accès mémoire
- *Représentation intermédiaire de haut niveau ou source*
  - ✓ Portabilité de la représentation intermédiaire
  - ✓ Sémantique du domaine embarquée dans le représentation intermédiaire

Multiplication des VMs

Le problème de la multiplication des VMs en système

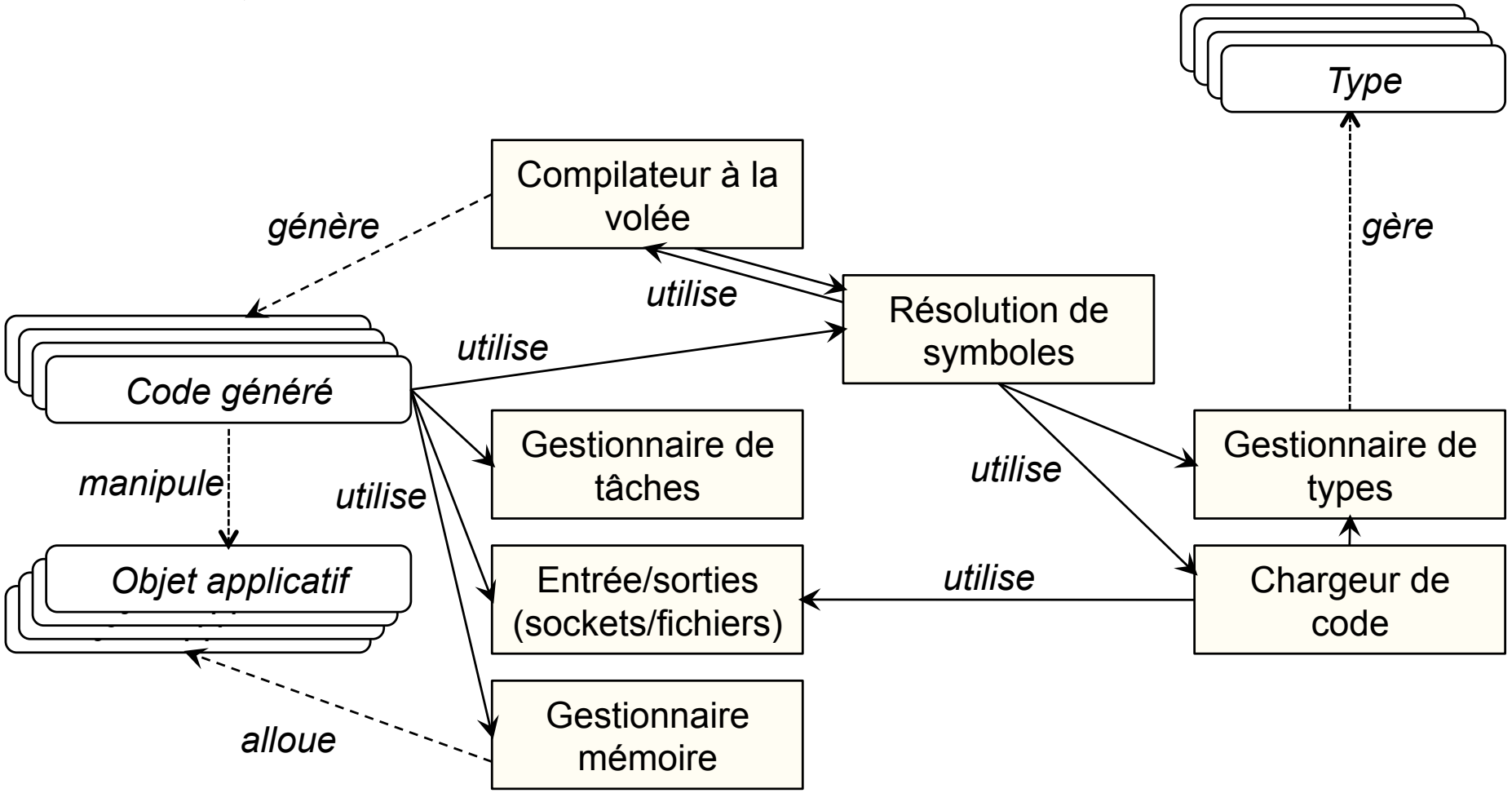
- ✓ **Communication locale** entre VMs lente
  - Pas de communication par mémoire partagée
  - Particulièrement sensible à l'heure des architectures massivement multicoeurs
- ✓ Gaspillage **mémoire et temps de calcul**
  - Pas de factorisation des bibliothèque standards, des composants communs du code généré par le compilateur à la volée

Le problème de la multiplication des VMs en génie logiciel

- ✓ **Temps de développement** et d'optimisation d'une VM long
  - Car VM = Système d'exploitation + Compilateur à la volée
- ✓ **Réutilisation** difficile des VMs existantes
  - VM très dépendante du langage d'entrée

Multiplication des VMs

Toutes les VMs suivent le même schéma



Multiplication des VMs

AutoVM : ***factorisation*** *des composants communs aux VMs dans un cœur* ***générique extensible***

- ✓ VMs chargées **en mémoire partagée**, communiquent par appel direct
  - Bonne performance de **communication**
- ✓ Factorisation des bibliothèques système et du code généré
  - Optimisation mémoire et temps de compilation

- ✓ Réutilisation des composants communs
  - Diminue le temps de développement d'une VM
- ✓ Extensibilité du cœur
  - Réutilisation/enrichissement du cœur

Multiplication des VMs

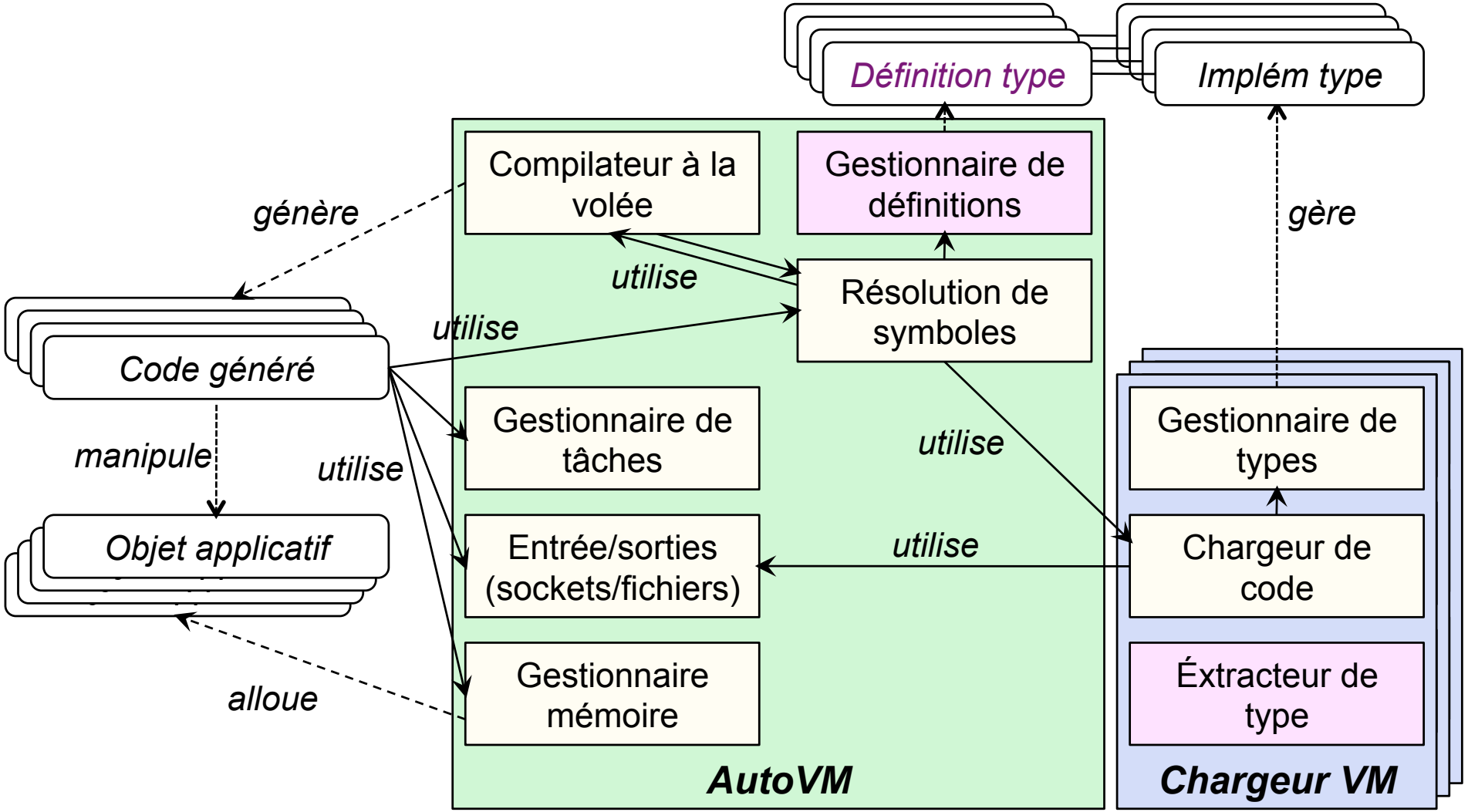
Objectif principale d'AutoVM :

Communication performante et interopérabilité entre VMs

- ✓ Plusieurs VMs dans le même espace d'adressage
- ✓ Un typage générique
- ✓ Une table des symboles générique

Multiplication des VMs

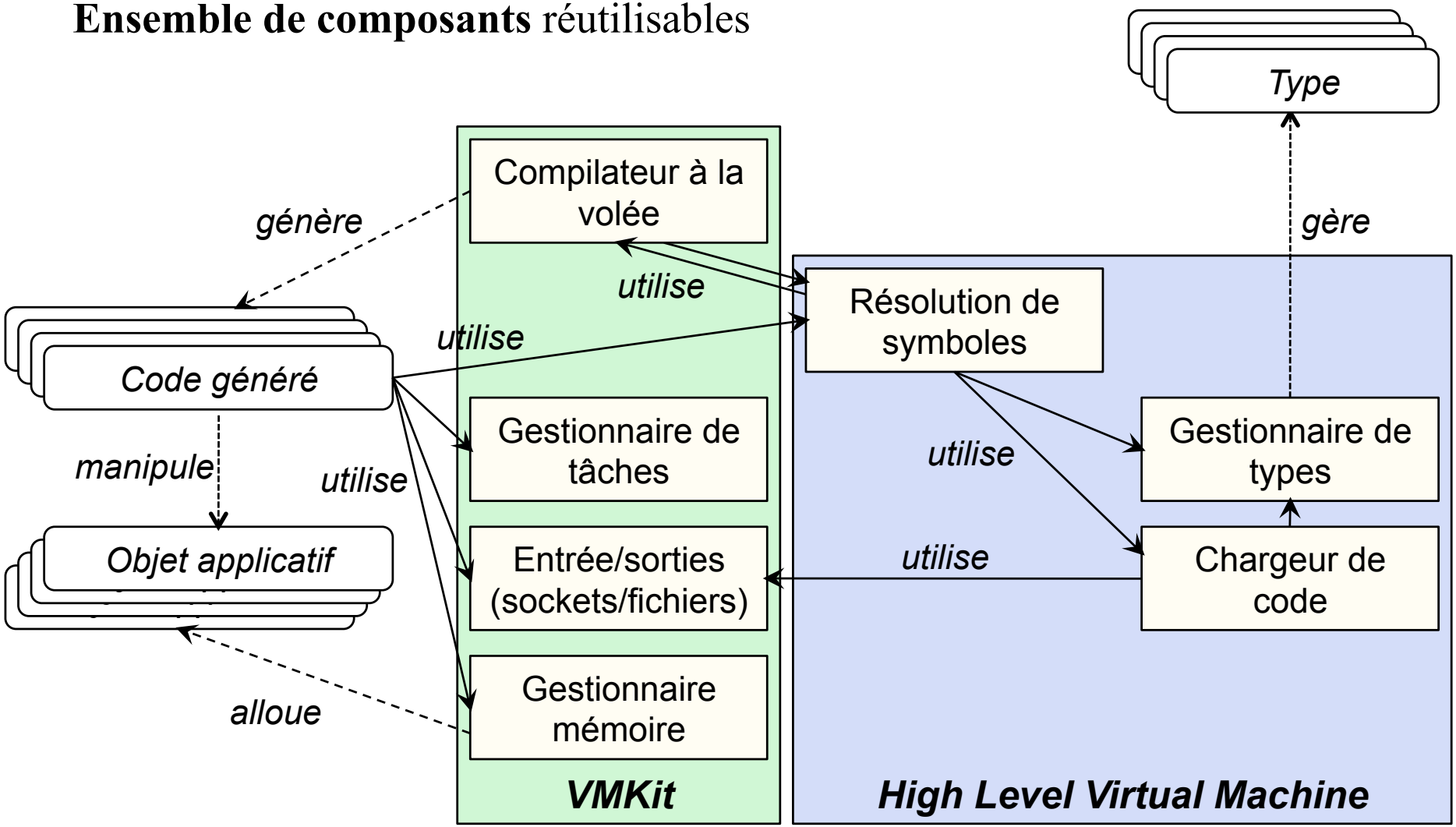
AutoVM : **Environnement d'exécution** extensible pour machine virtuelles



Multiplication des VMs

Travaux passés : VMKit (adresse uniquement les problèmes génie logiciel)

Ensemble de composants réutilisables



Multiplication des VMs

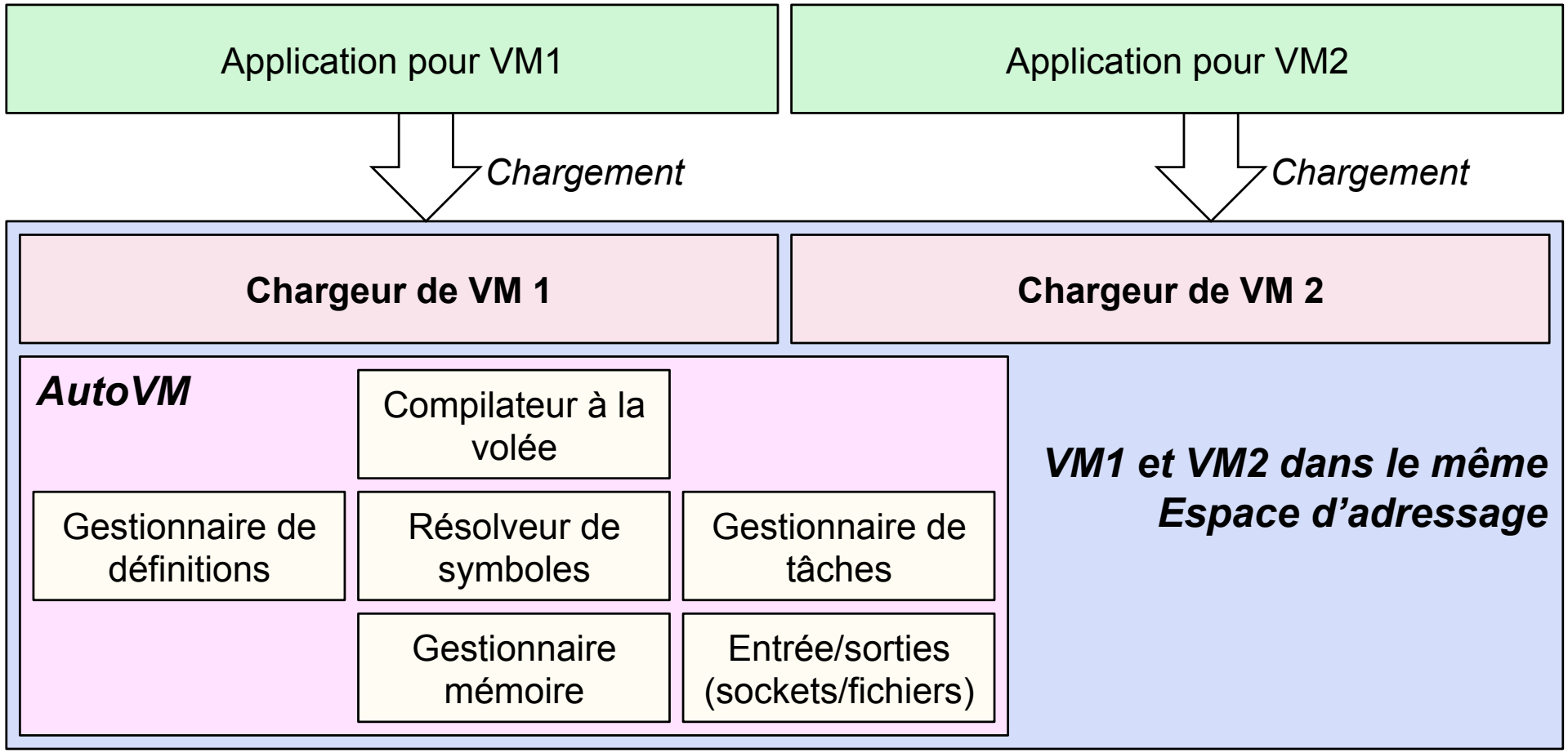
Chargeur de VM : construit une machine virtuelle complète

Composants d'un chargeur de VM

- ✓ Gestionnaire de types
  - Définit les types : classes, structures et espace de nommage
  - Définit la façon d'allouer les types (gc, pile, tas)
- ✓ Chargeur de code
  - Charge le code abstrait et construit les types
  - Compile le code abstrait via le composant « compilateur à la volée »
- ✓ Extracteur de types
  - Extrait des types AutoVM vers le code source d'une VM (équivalent de ce qui se fait avec WebServices via wsdl2java)

Multiplication des VMs

Exécution de plusieurs VMs dans AutoVM



Plan de la présentation

Partie I : Design d'AutoVM

1. Résolution de symboles et typage dans AutoVM
2. Génération de code dans AutoVM
3. Gestionnaire mémoire et génération de code

Partie II : Extensibilité dynamique et tour méta

4. Interopérabilité et extensibilité dynamique
5. Le langage AutoVM-L et le bootstrap

Partie III : conclusion

6. Conclusion

Résolution de symboles dans AutoVM

Résolution des symboles indépendamment des VMs

Laisse la liberté à la VM de gérer ses types en fonction de ses besoins

(héritage, transtypage, allocation gc/pile/tas)

Les symboles dans AutoVM :

Possède un nom, un type, un accès et un type conteneur

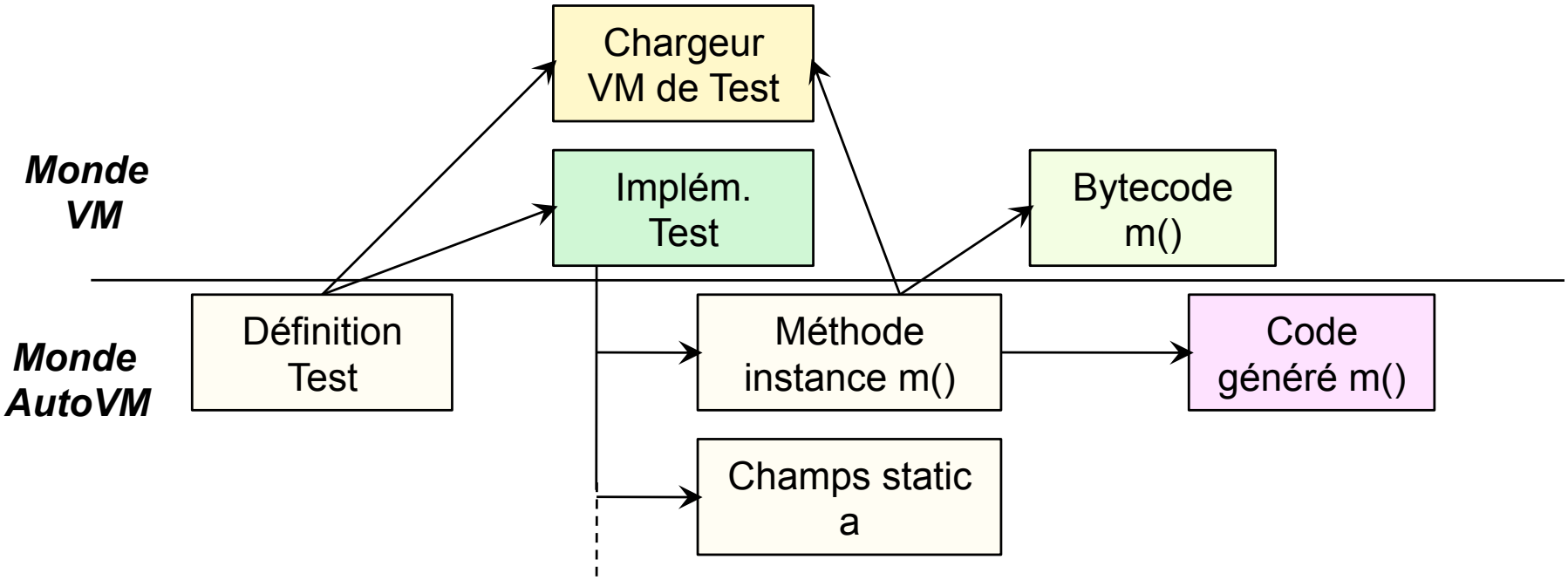
- ✓ Champs d'instance : offset par rapport au début de l'instance
- ✓ Champs statique : possède une valeur (primitif ou construit)
- ✓ Méthode d'instance : offset dans une table virtuelle + chargeur de VM + pointeur vers code généré + donnée opaque (source/bytecode)
- ✓ Méthode statique : chargeur de VM + pointeur vers code généré + donnée opaque (source/bytecode)
- ✓ Mot clé : méthode statique qui s'exécute lors de la compilation

Résolution de symboles dans AutoVM

Typage générique dans AutoVM (1300 loc)

- ✓ Type primitif : octet, mot, flottant, booléen… (types primitifs IDL Corba)
- ✓ Type construit = définition de type :
  Possède une implémentation et un chargeur de VM

  Le chargeur s'occupe de construire l'implémentation du type



Résolution de symboles dans AutoVM

Interface d'une implémentation de type

- ✓ **Recherche de membre** à partir d'un nom, d'un type et d'une sémantique (champs, méthode, mot-clé, instance, statique)

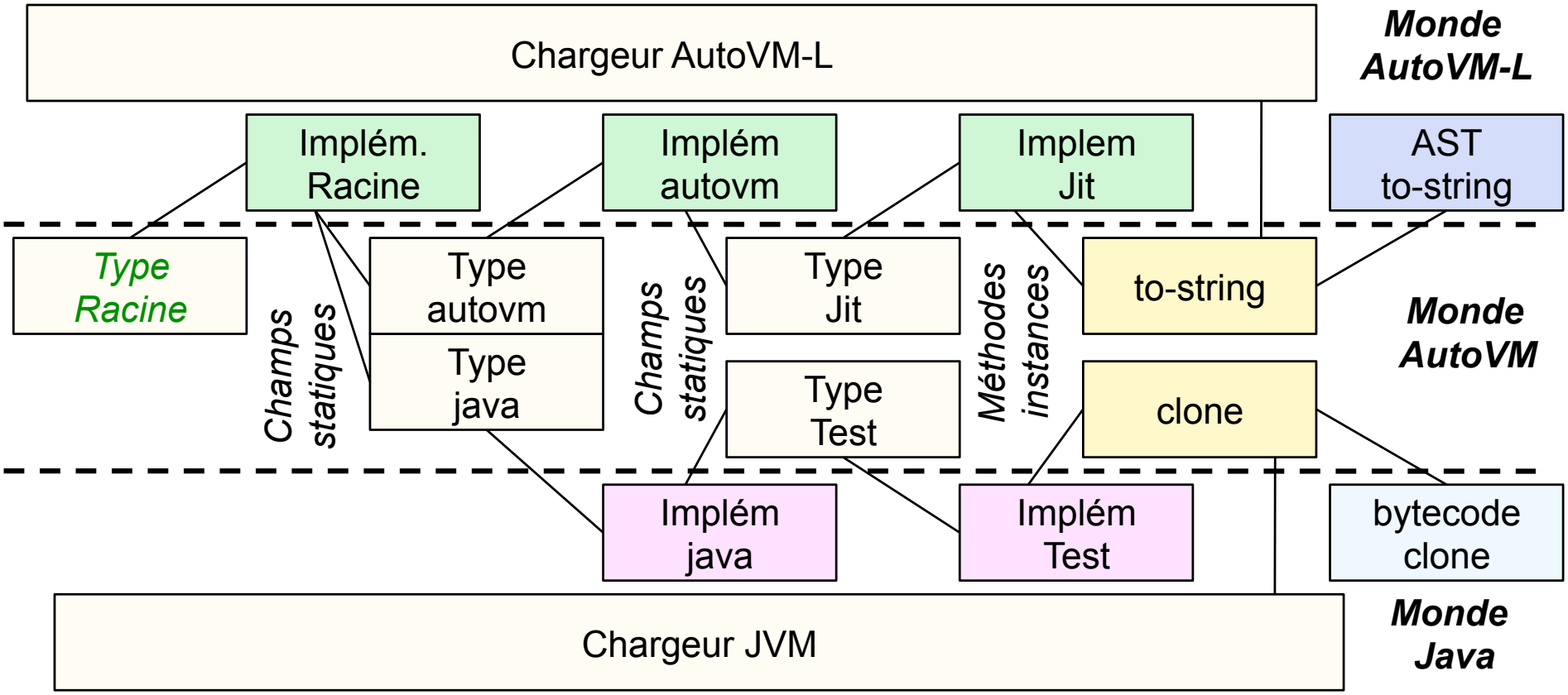
- ✓ Test de compatibilité de type (instanceOf)
- ✓ Allocation d'une instance
- ✓ Duplication d'une instance
- ✓ Opérateur de transtypage vers un des parents
- ✓ Consultation de la table virtuelle des instances
- ✓ Consultation du nombre d'octets des instances

Résolution de symboles dans AutoVM

Résolution de symboles générique dans AutoVM :

un type appelé racine *: la table des symboles*

Si un champs statique est un type, c'est un sous-module



Plan de la présentation

Partie I : Design d'AutoVM

1. Résolution de symboles et typage dans AutoVM
2. Génération de code dans AutoVM
3. Gestionnaire mémoire et génération de code

Partie II : Extensibilité dynamique et tour méta

4. Interopérabilité et extensibilité dynamique
5. Le langage AutoVM-L et le bootstrap

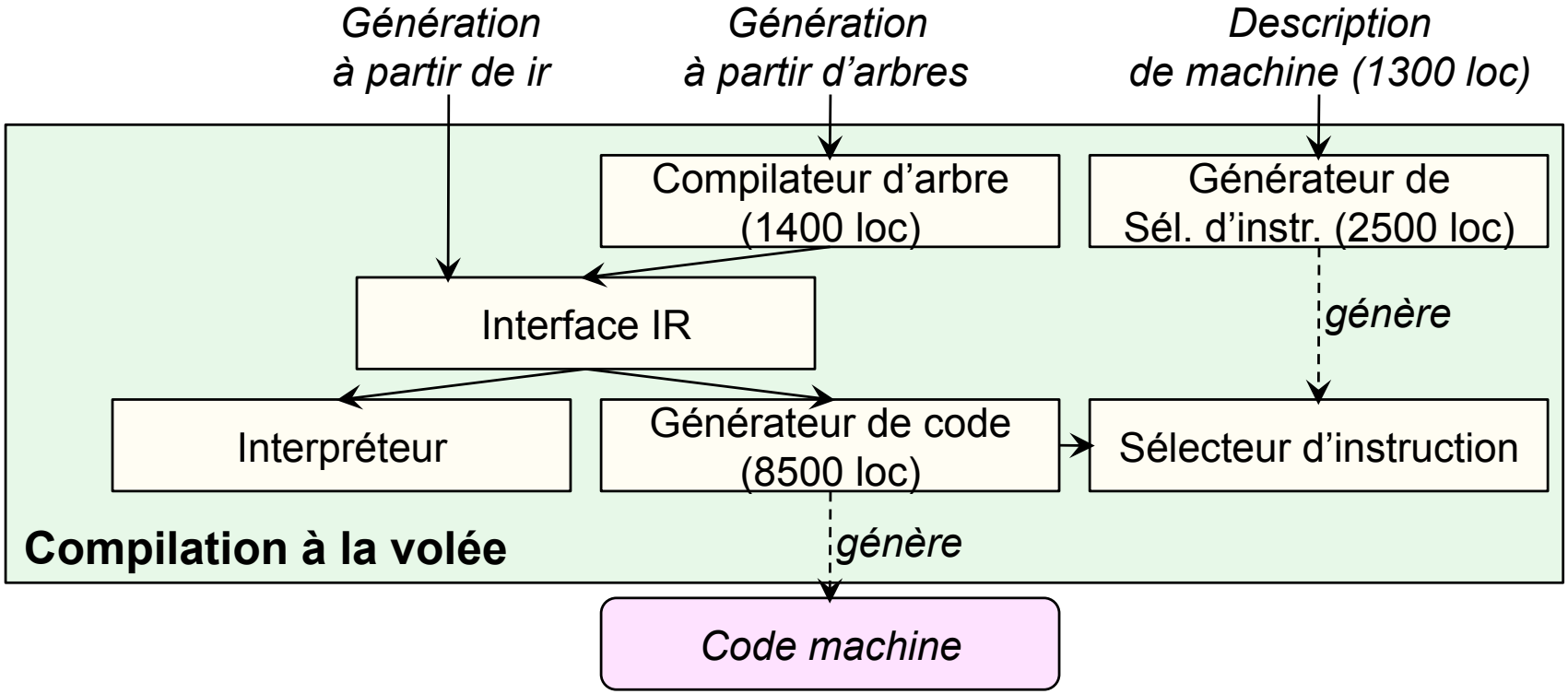
Partie III : conclusion

6. Conclusion

Génération de code dans AutoVM

Deux niveaux de génération de code

- ✓ Génération de code à partir d'un arbre de syntaxe abstraite (AST)
  (VM qui chargent un code source)
- ✓ Génération de code à partir d'une représentation intermédiaire
  (VM qui chargent un bytecode)



Génération de code dans AutoVM

Le compilateur d'arbres

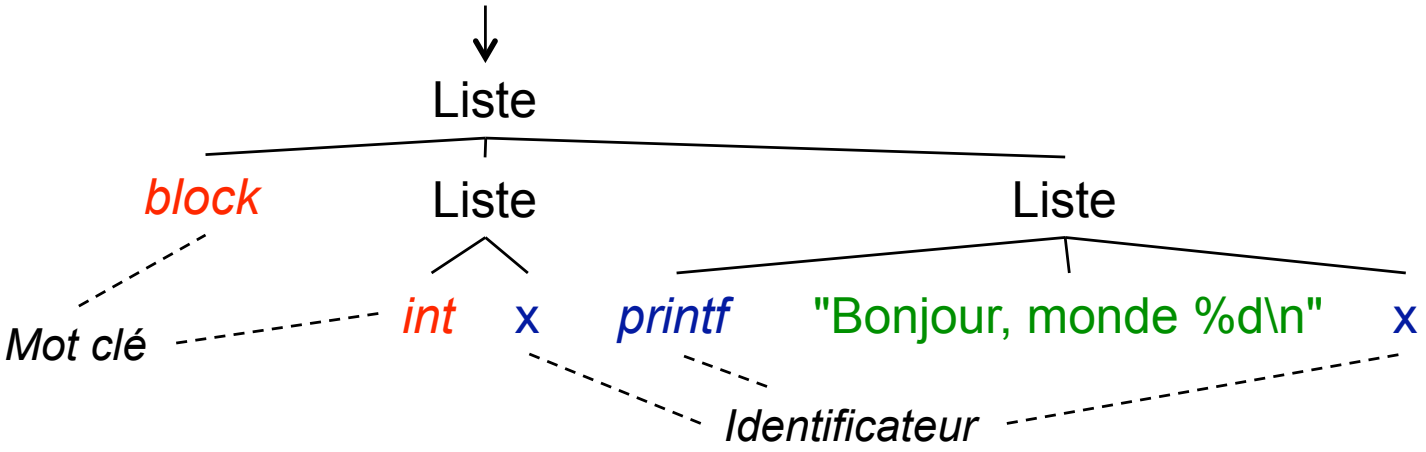
- ✓ Valeur directe : instance, entier, flottant, …
- ✓ Identificateur : liste de chaînes de caractères (à partir de la racine)
- ✓ Liste d'éléments : application fonctionnelle

  Si premier élément est un mot clé ⇒ compilation déléguée à la fonction mot-clé

  Sinon appel de méthode ou consultation de champs

Tout langage peut se transformer en AST AutoVM

```
{ int x; printf("Bonjour, monde %d\n",x); }
```



Génération de code dans AutoVM

La représentation intermédiaire

- ✓ Opération arithmétiques
- ✓ Saut direct/indirect/conditionnel
- ✓ Retour de fonction
- ✓ Appel de méthode statique/d'instance/d'interface
- ✓ Consultation de champs statique/d'instance
- ✓ Lecture/écriture aux tableaux
- ✓ Gestion d'exceptions
- ✓ Allocation d'objets
- ✓ Définition/lecture/écriture paramètres et variables locales

- ✓ Possibilité d'ajouter de nouvelles primitives avec la VM
  (par exemple, mot clé pour les appels inter-segments ia32)

Génération de code dans AutoVM

Rôle de l'interpréteur

Si la méthode est un point chaud, passage au générateur de code

- ✓ Vérifier que le typage est consistent
- ✓ Inférer les types manquants
- ✓ Exécuter le code

Rôle du générateur de code

Si la méthode est un point très chaud, re-génération avec optimisation

- ✓ Vérifier que le typage est consistent
- ✓ Inférer les types manquants
- ✓ Appliquer des optimisations (si point très chaud)
- ✓ Générer du code binaire correspondant à la représentation intermédiaire

Génération de code dans AutoVM

Le générateur de code

Cinq étapes dont quatre obligatoires

- ✓ **Construction des blocs de base** et de la séquence d'instruction

  Inférence de type en utilisant le typage générique

- ✓ Passage en **forme SSA et optimisation** (optionnel, réglable)

  Sparse Conditionnal Constant Propagation, Global Value Numbering, Global Code Motion, Loop unrolling, Trace scheduler, Array bound check elimination, suppression des blocs inutilisés

- ✓ **Sélection des instructions de la machine**

  Génération du reconnaisseurs de tuiles à partir d'une description de haut niveau de la machine (programmation dynamique)

- ✓ **Allocation de registres**

  Variante de linear scan

- ✓ **Émission du code** en mémoire

Plan de la présentation

Partie I : Design d'AutoVM

1. Résolution de symboles et typage dans AutoVM
2. Génération de code dans AutoVM
3. Gestionnaire mémoire et génération de code

Partie II : Extensibilité dynamique et tour méta

4. Interopérabilité et extensibilité dynamique
5. Le langage AutoVM-L et le bootstrap

Partie III : conclusion

6. Conclusion

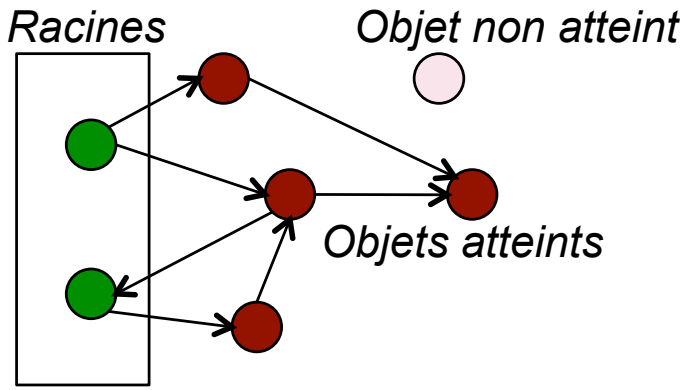
Gestionnaire mémoire et génération de code

Gestionnaire mémoire (pas encore réalisé)

- ✓ Un allocateur mémoire
- ✓ Un récupérateur mémoire (ramasse-miettes)

L'algorithme classique de récupérateur mémoire : le *mark-and-sweep*

- ✓ Sélection des objets racines (pile d'exécution et racine table des symboles)
- ✓ Parcourt des objets atteignables à partir de la racine
- ✓ Tout objet non-atteint peut être récupéré



Gestionnaire mémoire et génération de code

1. Algorithmes exacts : différencier les entiers des références
   - ✓ Typage donne le contenu d'un objet
   - ✓ Nécessite aide du JIT pour la pile d'exécution : *carte de la pile* (StackMap)

2. Algorithmes générationnels : ne parcourir que les objets récemment alloués
   - ✓ Racines sont piles + objets vieux qui référencent des jeunes
   - ✓ Nécessite de repérer les racines : *barrière en écriture*

3. Algorithmes incrémentaux : ne pas bloquer l'application pendant la collection
   - ✓ Nécessite de repérer les mutations du graphe pendant le parcours : *barrière en écriture*

4. Algorithmes compactant : déplace les objets pendant collection
   - ✓ Nécessite algorithme exact et donc *carte de la pile*

Gestionnaire mémoire et génération de code

Génération de la carte de la pile :

Toute variable locale vit en une et une seule position aux points de contrôle

- ✓ Position : dans la pile ou dans un registre
- ✓ Point de contrôle : arc arrière de boucle non bornée, retour de fonction

Génération de la barrière en écriture

Appel au gestionnaire mémoire pour insérer la barrière

- ✓ Insertion en ligne de la représentation intermédiaire *(après inférence de type)*

Génération de point de collection (handshake) aux points de contrôle

Plan de la présentation

Partie I : Design d'AutoVM

1. Résolution de symboles et typage dans AutoVM
2. Génération de code dans AutoVM
3. Gestionnaire mémoire et génération de code

Partie II : Extensibilité dynamique et tour méta

4. Interopérabilité et extensibilité dynamique
5. Le langage AutoVM-L et le bootstrap

Partie III : conclusion

6. Conclusion

Interopérabilité et extensibilité dynamique

Un chargeur de VM doit interagir avec AutoVM

- ✓ Chargeur de VM vers AutoVM : réutilisation des composants communs
- ✓ AutoVM vers chargeur de VM : chargement dynamique d'application…

Une VM doit interagir avec son application

- ✓ VM vers applications : appel de la méthode principale
- ✓ Applications vers VM : réflexion, méthodes natives

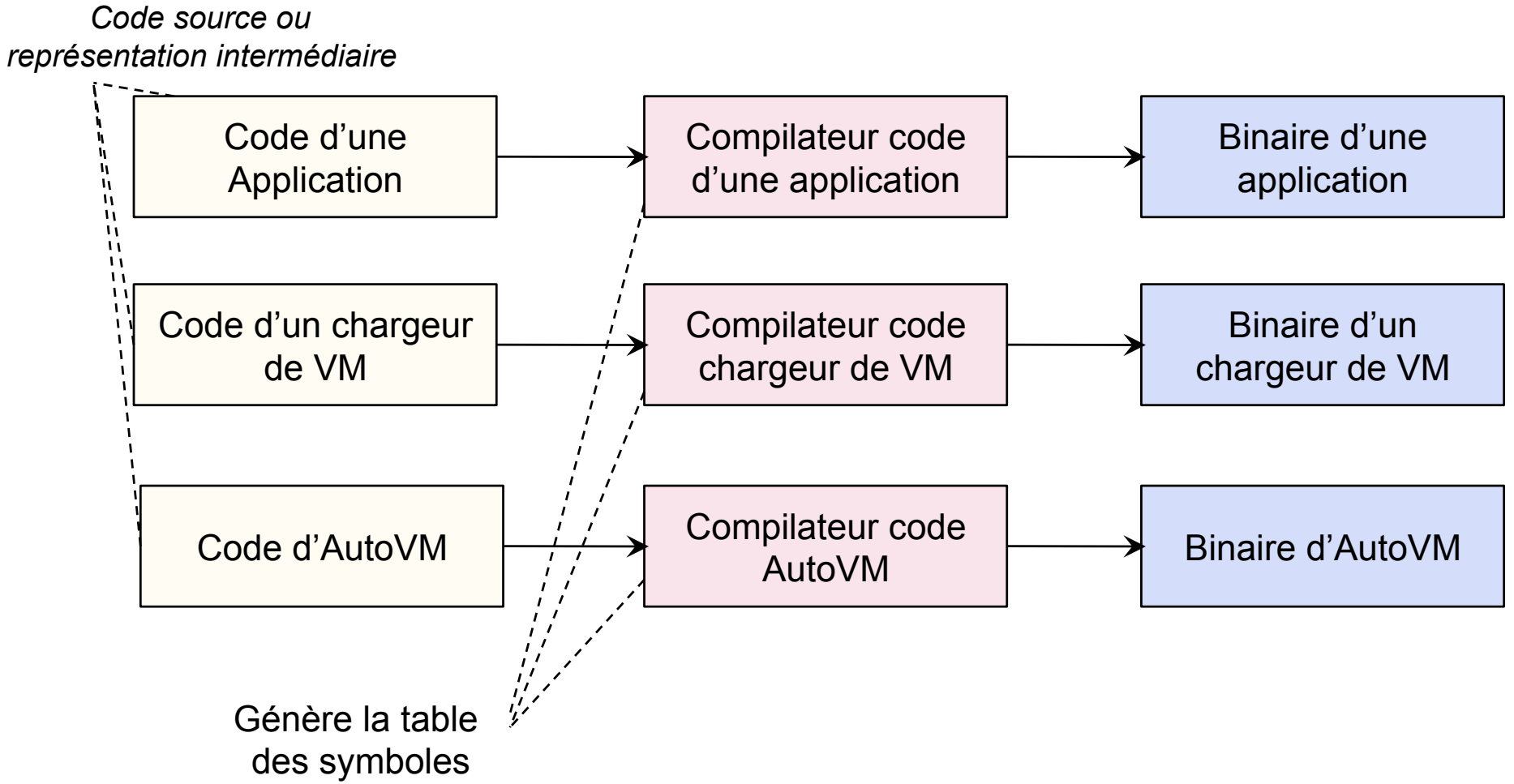
Les applications doivent interagir entre elles

- ✓ Applications vers applications : appel de procédure inter-VMs

⇒ **Table des symboles unifiée** de AutoVM, chargeurs de VM, applications

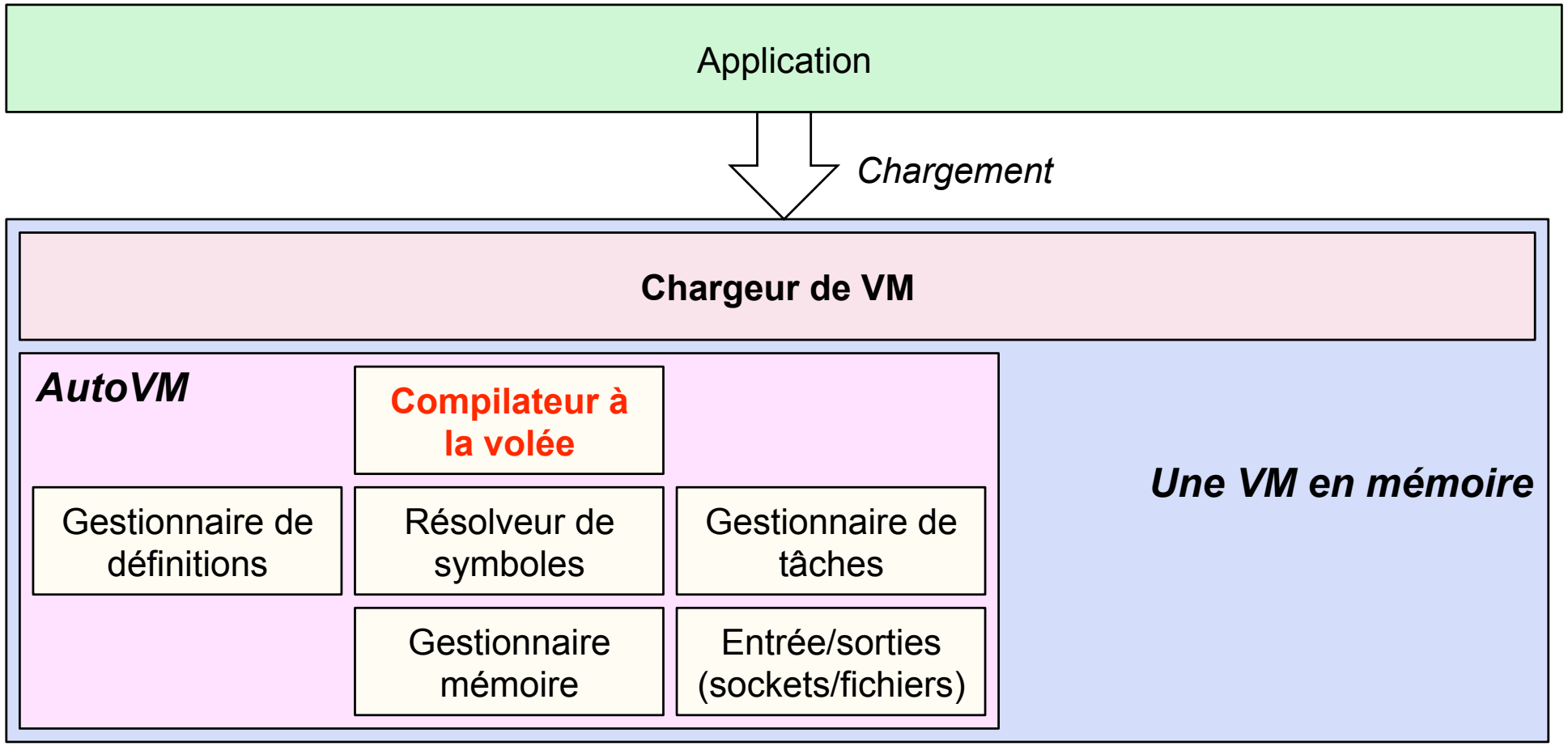
Interopérabilité et extensibilité dynamique

Table des symboles unifiée ⇒ unification des compilateurs



Interopérabilité et extensibilité dynamique

Mais compilateur des applications imposé par le design



Interopérabilité et extensibilité dynamique

Seule solution :

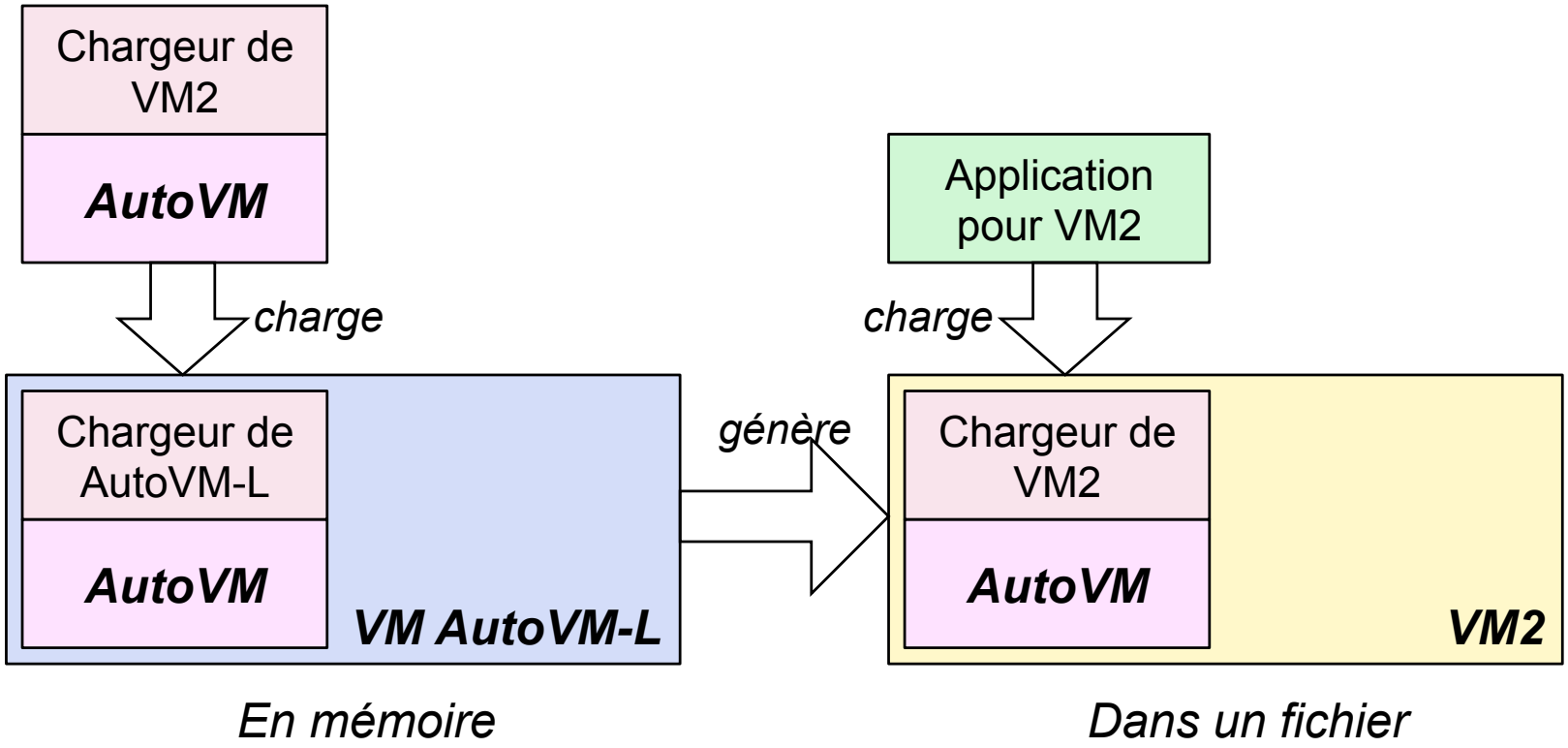
faire compiler AutoVM et les chargeurs de VM par AutoVM

- ✓ AutoVM écrite dans un langage **AutoVM-L**
- ✓ Un Chargeur de code pour AutoVM-L définit une VM pour AutoVM-L (1300 loc)
- \+ Compilateur IR doit générer du *code statique et dynamique*

Interopérabilité et extensibilité dynamique

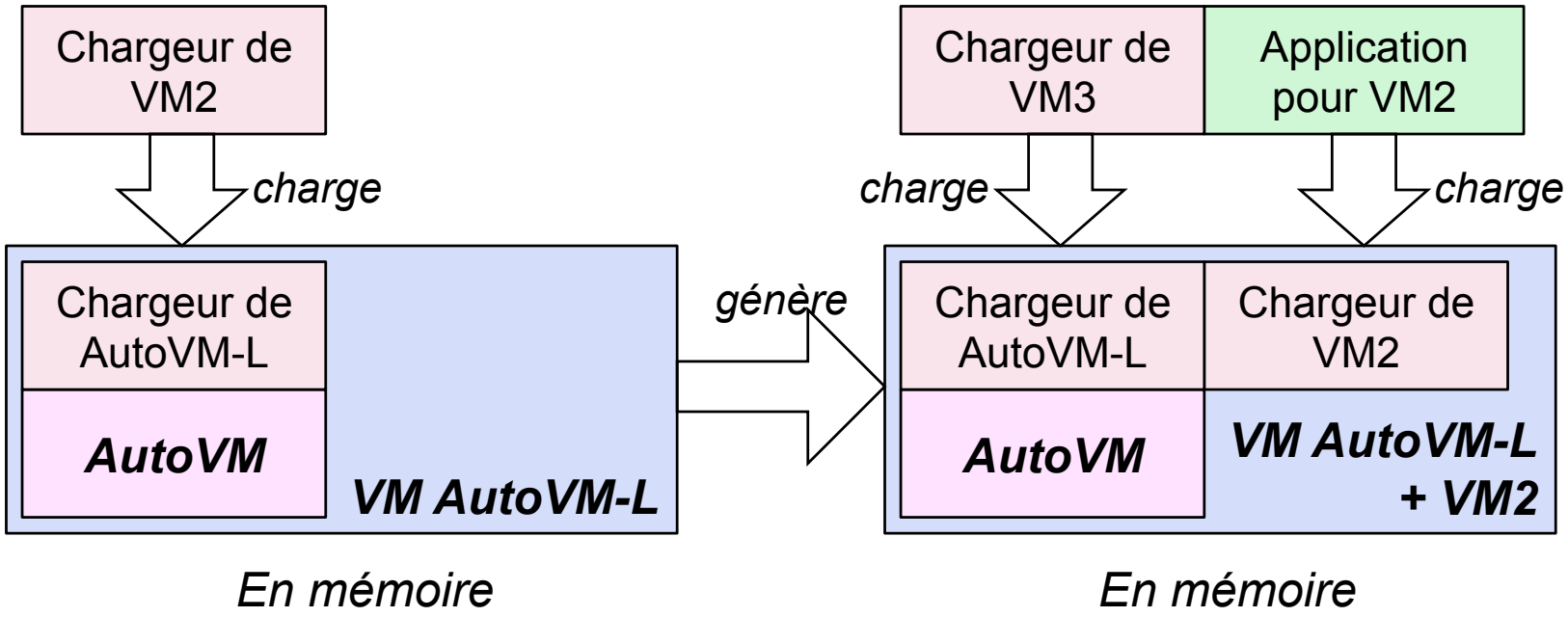
Schéma de Compilation statique

(chargeur de VM2 peut être le chargeur d'AutoVM-L)



Interopérabilité et extensibilité dynamique

Schéma de compilation dynamique et extensibilité



Interopérabilité et extensibilité dynamique

Corolaires

Un chargeur de VM doit être écrit dans un langage qui possède déjà un chargeur

(exemple précédent : chargeur VM2 écrit dans le langage AutoVM-L)

Problème : et le chargeur d'AutoVM-L!

Plan de la présentation

Partie I : Design d'AutoVM

1. Résolution de symboles et typage dans AutoVM
2. Génération de code dans AutoVM
3. Gestionnaire mémoire et génération de code

Partie II : Extensibilité dynamique et tour méta

4. Interopérabilité et extensibilité dynamique
5. Le langage AutoVM-L et le bootstrap

Partie III : conclusion

6. Conclusion

Le langage AutoVM-L et le bootstrap

Choix du langage de conception d'AutoVM : AutoVM-L

Pré requis 1 : ***langage spécialisé dans la compilation***

- ✓ Texte proche des arbres de syntaxe

Pré requis 2 : ***définition aisée de nouveaux mots clés***

pour le JIT, pour lexer/parser, ajout de nouveaux mots clés pour les VMs

- ✓ Système de macros/syntaxes avancé

AutoVM-L : Scheme minimal

- ✓ Syntaxe : Scheme
- ✓ Sémantique : impératif + objet (≠ scheme)

(define-syntax, lambda, define, define-class, let, while, new, set! …)

Le langage AutoVM-L et le bootstrap

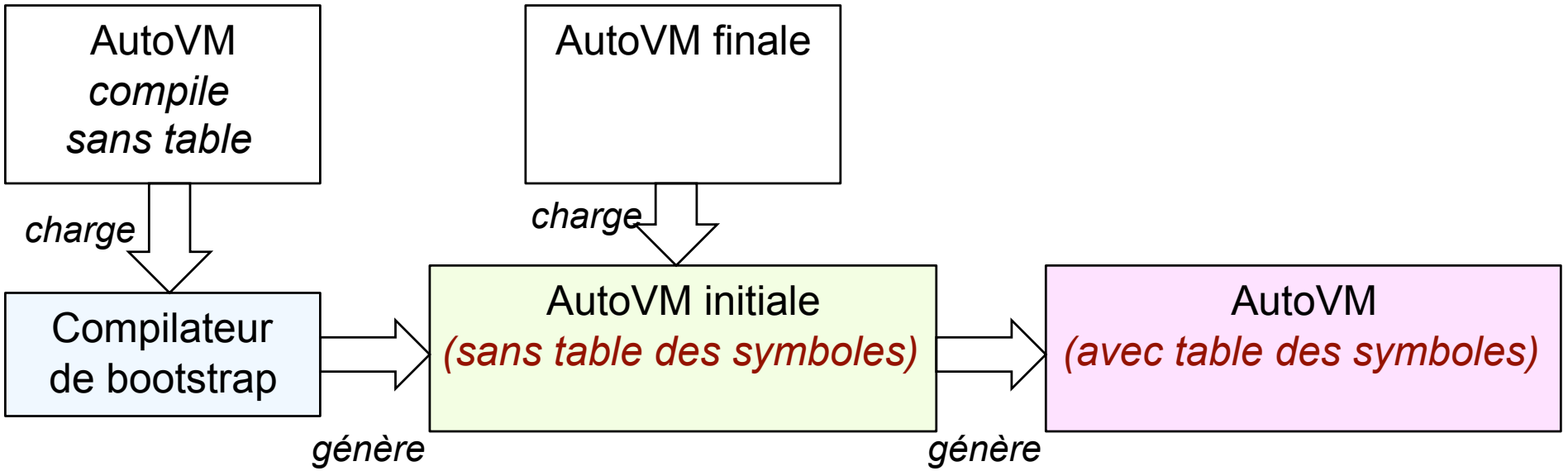
Problème du bootstrap : AutoVM doit être compilé par AutoVM pour fonctionner

(pour avoir sa table des symboles)

Schéma classique de bootstrap avec deux versions d'AutoVM

- ✓ Première : capable de compiler le code sans table des symboles d'AutoVM
- ✓ Seconde : version normale

Difficile car AutoVM = 27'500 loc aujourd'hui…



Le langage AutoVM-L et le bootstrap

Principe : enrichir le compilateur de bootstrap pour qu'il génère la table des symboles de AutoVM

Réutilisation d'un compilateur de Scheme de l'équipe : Microvm

- ✓ Langage impératif, pas de structure, pas orienté objet
- ✓ Possibilité d'ajouter de nouveaux mots clés

Ajout d'un nouvel ensemble de mots clés

- ✓ Orientés objets : define-class, new,…
- ✓ *Génère la table des symboles AutoVM en utilisant types AutoVM (la classe de Class doit être le classe Class)*
- – Contrepartie : ces mots clés sont interprétés…

Le langage AutoVM-L et le bootstrap

Utilisation de code AutoVM-L pour construire l'AutoVM de bootstrap

Partie AutoVM

- ✓ *Langage objet :* define-class, new, instance-of?, tableaux…
- ✓ *Résolveur de symboles* : symboles et types d'AutoVM
- ✓ *Gestionnaire de définitions* : gestion des définitions de types

Partie chargeur de AutoVM-L

- ✓ *Gestionnaire de types* : définition du format des types internes de AutoVM-L

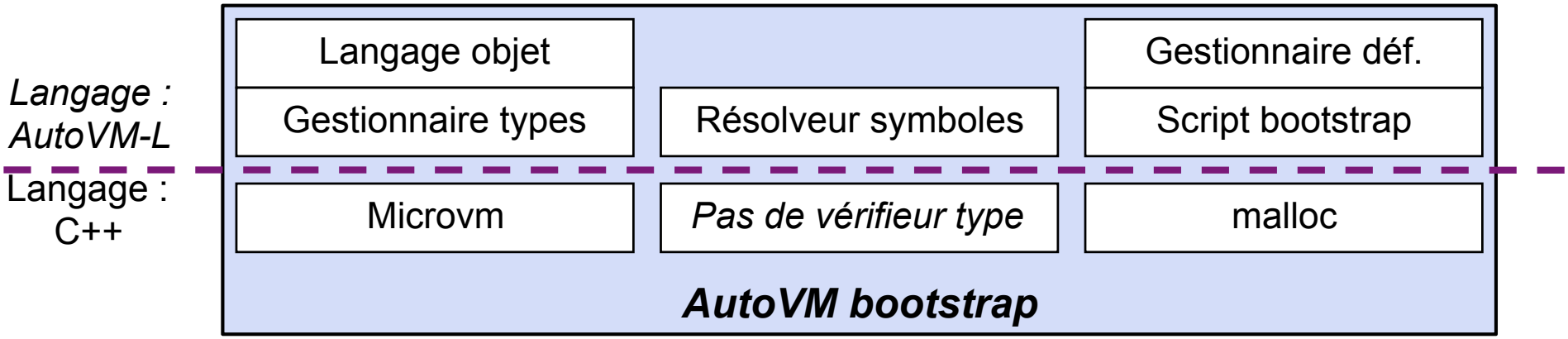
Partie Script de bootstrap *(1000 loc)* :

- ✧ Charge les composants précédents
- ✧ Construit les types de base à partir des types de base

Le langage AutoVM-L et le bootstrap

Bootstrap étape 1 : Construction d'un interpréteur AutoVM-L

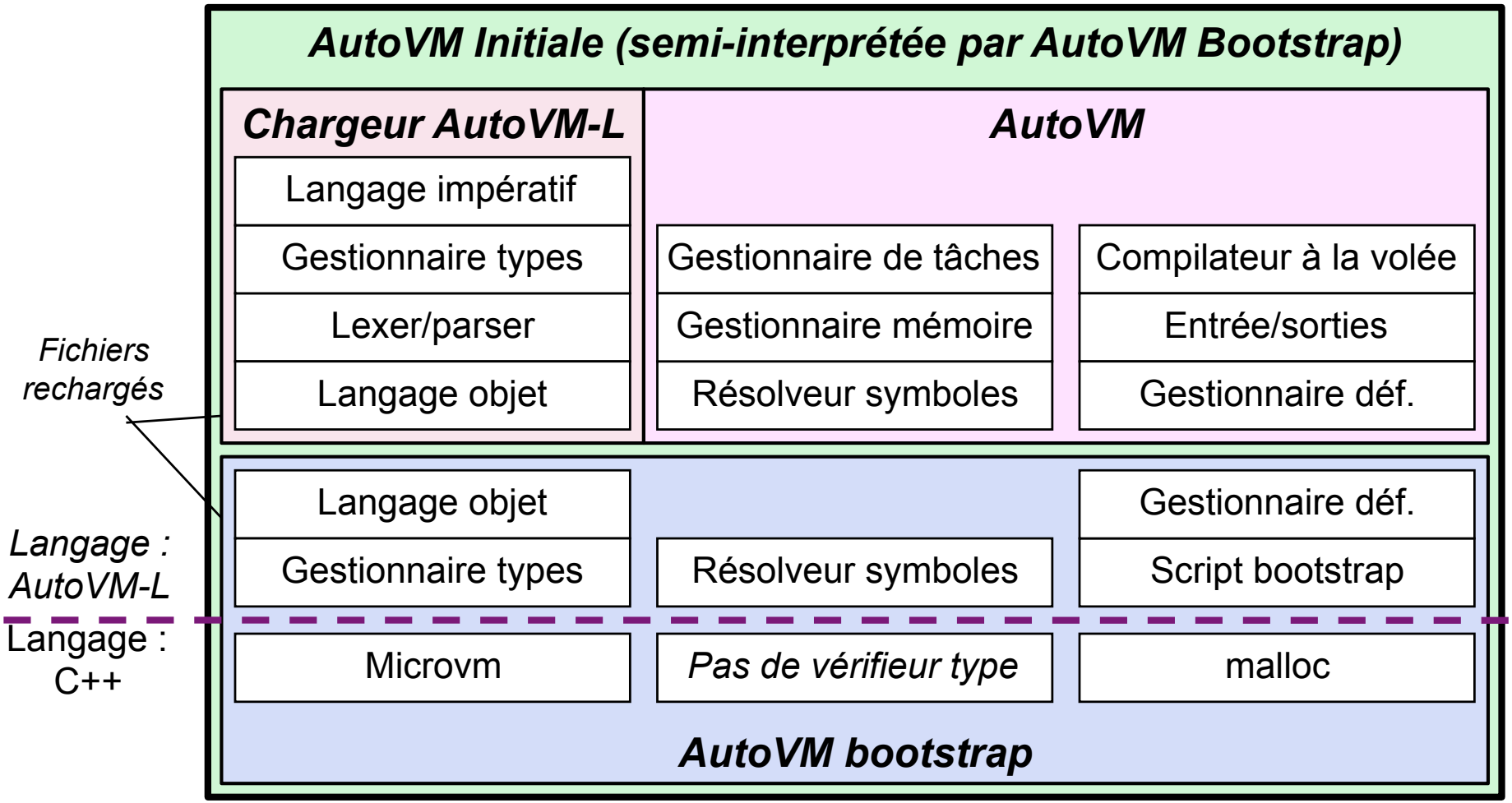
- ✓ Capable de charger du AutoVM-L, code semi-interprété



Le langage AutoVM-L et le bootstrap

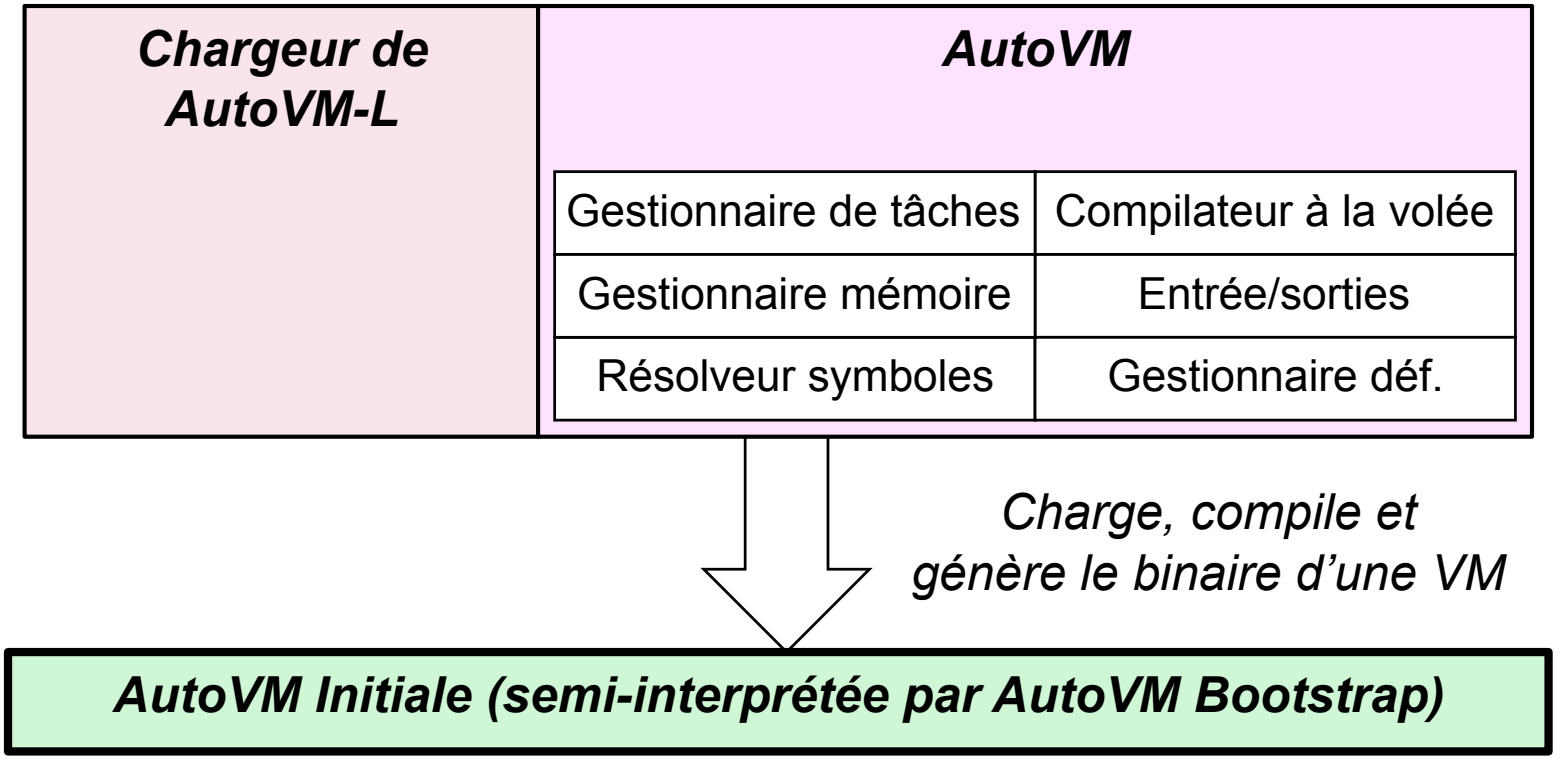
Bootstrap étape 2 : Génération d'une AutoVM initiale en mémoire

- ✓ Capable de charger AutoVM-L, code compilé par compilateur à la volée



Le langage AutoVM-L et le bootstrap

Bootstrap étape 3 : Génération de AutoVM



Plan de la présentation

Partie I : Design d'AutoVM

1. Résolution de symboles et typage dans AutoVM
2. Génération de code dans AutoVM
3. Gestionnaire mémoire et génération de code

Partie II : Extensibilité dynamique et tour méta

4. Interopérabilité et extensibilité dynamique
5. Le langage AutoVM-L et le bootstrap

Partie III : conclusion

6. Conclusion

Conclusion

AutoVM : cœur extensible de machine virtuelle

- ✓ Communication performante entre VMs
- ✓ Factorisation des VMs et du code compilé
- ✓ Réutilisation des composants internes pour développer rapidement des VMs

Encore un peu d'ingénierie

- ✓ Pour terminer AutoVM (actuellement 27500 loc)
  - Complétude du compilateur à la volée, écriture de l'interpréteur
  - Génération de fichiers binaire à partir de la mémoire
  - Ramasse-miettes
- ✓ Développer des chargeurs de VM (C, JVM, .Net, autre)

Expériences de recherche à partir d'AutoVM

- ✓ Embarquer AutoVM dans un noyau Linux pour charger de façon sûr des modules
- ✓ Expérimentation sur des ramasse-miettes multi-cœurs
- ✓ Vers un langage dédié à la construction de machines virtuelles