Langages Fonctionnels, Typage et Interopérabilité

O'Caml sur .NET

Raphaël Montelatici

Equipe PPS (UMR 7126) - Université Denis Diderot (Paris VII)

Soutenance de thèse - 15/03/2007

Plan

- Introduction : la plate-forme .NET et les objectifs du projet OCamIL
- Compilation : typage et représentation des valeurs
  - reconstruction ou propagation des types
  - implantations des types structurés, fermetures, exceptions . . .
- Interopérabilité
  $\Rightarrow$ de l'interface externe de bas niveau à O'Jacaré.Net
- Réalisations et applications
  - autour du compilateur : bootstrap et toplevel
  - sérialisation sûre, exemple d'appliquette
- Travaux futurs et conclusion

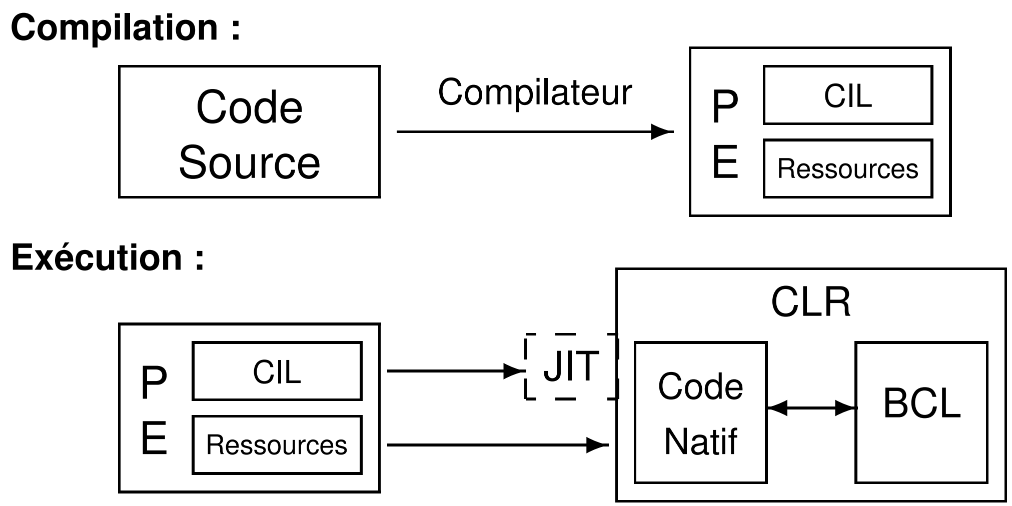
Plate-forme .NET et projet OCamIL

Plate-forme .NET (1/3)

Machine virtuelle à pile CLR :

- fondée sur un système de classes CTS
  - typage dynamique
  - chargeur de classes + versionnement / signature
- exécutant du code-octet géré CIL
  - manipulation des types de base
  - appel de fonctions
  - appel de méthodes avec liaison tardive
  - exceptions

Plate-forme .NET (2/3)



Catégories de code exécuté :

- natif
- géré (code-octet CIL)
- géré vérifiable (type-safe)

Compilation JIT *(Just In Time)*

Plate-forme .NET (3/3)

- Services de la plate-forme :
  - gestion de la mémoire (GC)
  - threading
  - chargement dynamique
  - introspection
  - GAC (global assembly cache)
- Interopérabilité :
  - cherche à héberger des langages variés
    - CLR = Common Language Runtime
    - CTS = Common Type System
    - CIL = Common Intermediate Language
  - interfaces avec le monde natif (COM, P/Invoke)

.NET : implantations et langages

Implantations :

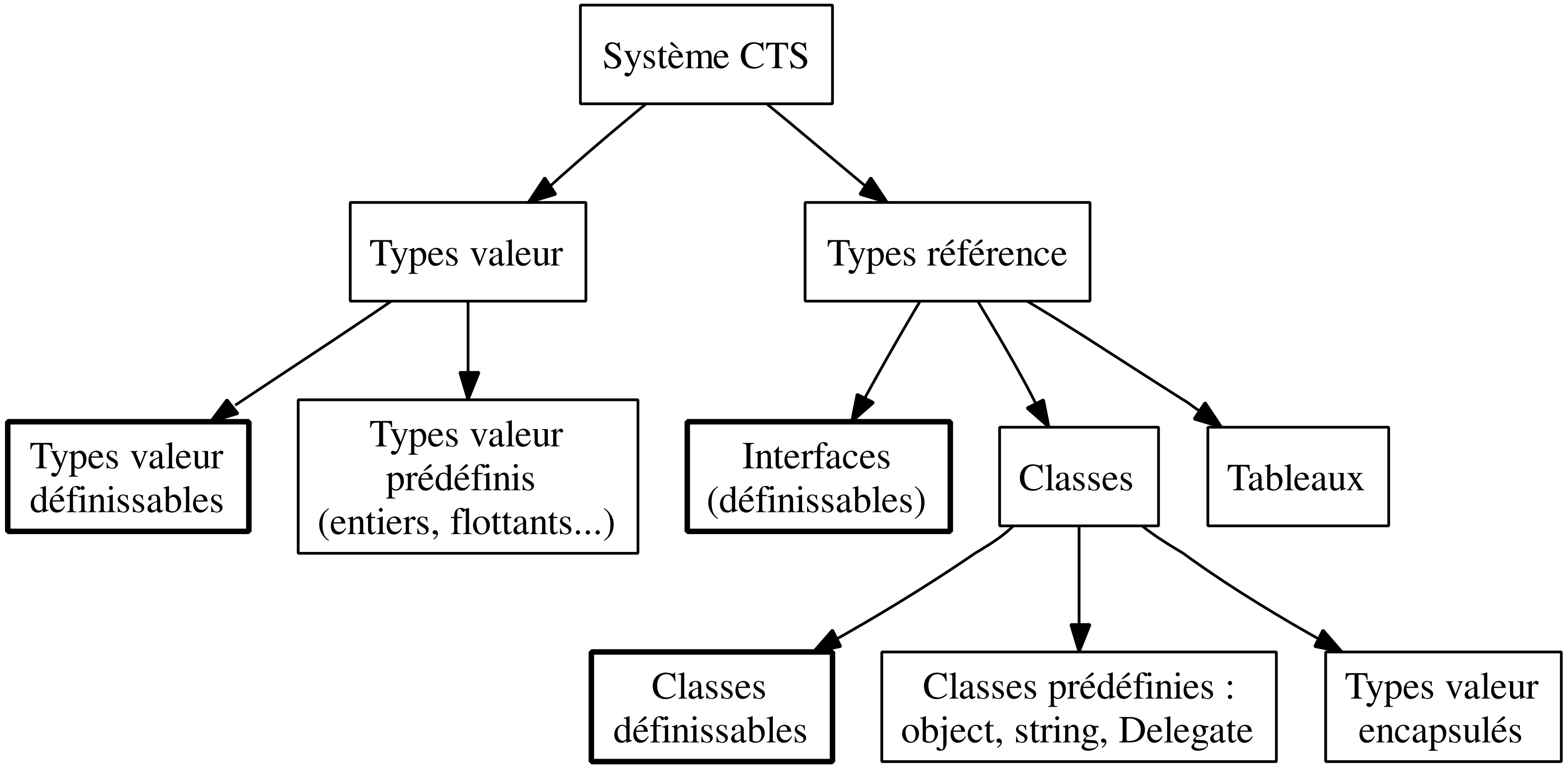
- .NET Framework (Windows seulement)
- Rotor
- Mono
- Portable.NET

Langages :

- C#, Visual Basic.Net, Visual C++, J#
- Eiffel, A#, Mercury . . .
- Haskell, Sml, Scheme, Bigloo, F#, Nemerle . . .

Pas aussi portable que Java mais multi-langages.

Le système de types CTS



- Types référence et types valeur
- Typage dynamique : cast sûr, introspection, sérialisation sûre

La famille ML sur .NET

- F#
  - Seulement un dialecte de Caml-Light : pas de module ni d'objet O'Caml
  - Incorporation du modèle objet du CTS
  - Efficacité correcte
- SML.NET
  - Bonne compatibilité avec SML
  - Incorporation du modèle objet du CTS
  - Pas de toplevel
  - Efficace mais pas de compilation séparée (monomorphisation globale)

Le projet OCamIL

Motivations :

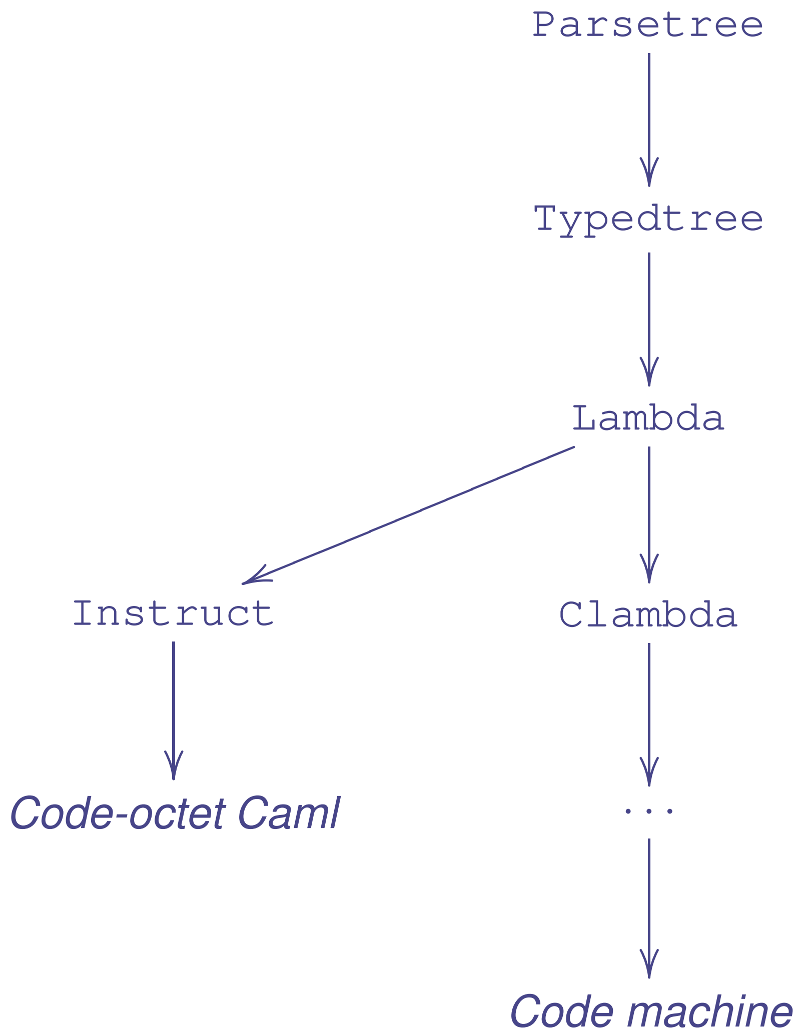
- Porter l'ensemble du langage sur la nouvelle plate-forme.
- Tirer profit de cette plate-forme : outils et interopérabilité.
- Tester la plate-forme sur un projet libre et indépendant.

Contraintes :

- Compatibilité maximale
- Pouvoir suivre l'évolution du compilateur INRIA
- Pouvoir interopérer
- Émettre du code géré vérifiable
- Essayer de garder une bonne efficacité

Compilation : typage et représentation des valeurs

Architecture de O'Caml



Le langage intermédiaire CLambda

```
type ulambda =
    Uvar of Ident.t
  | Ulet of Ident.t * ulambda * ulambda
  | Udirect_apply of function_label * ulambda list
  | Ugeneric_apply of ulambda * ulambda list
  | Uclosure of
      (function_label * int * Ident.t list * ulambda) list
              * ulambda list
  | Uoffset of ulambda * int
  | Uprim of primitive * ulambda list
  | ...
```

λ-calcul avec gestion explicite de fermetures, structures de contrôle et primitives.

Typage

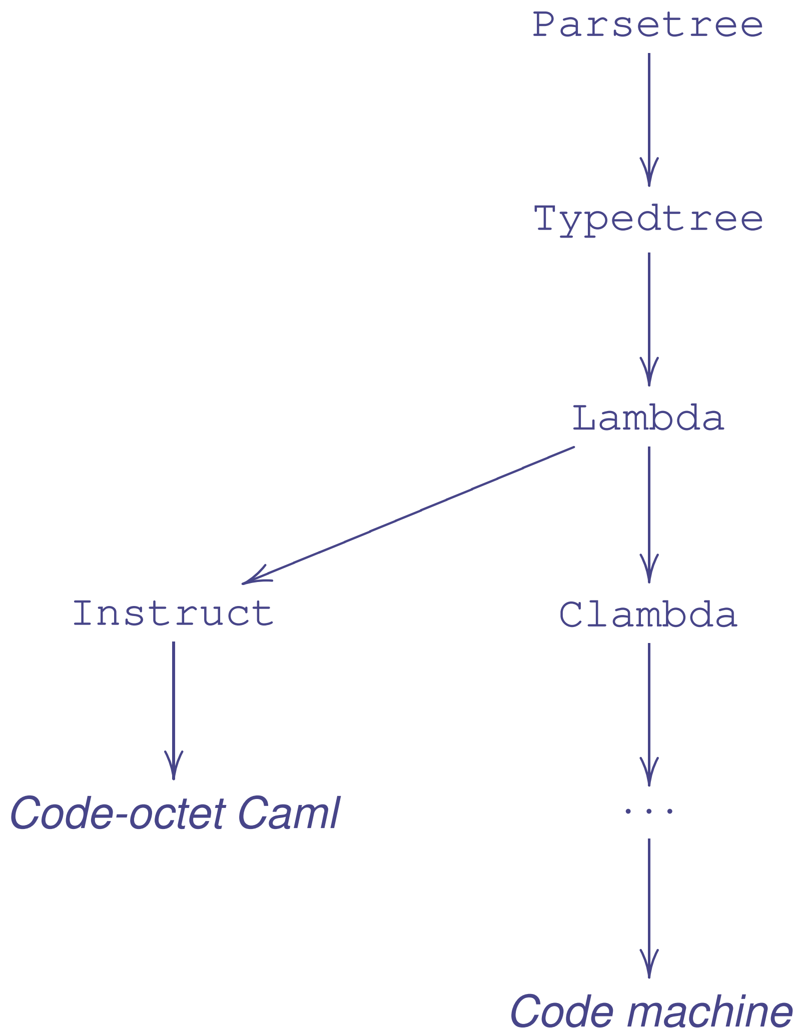
Problème : produire du code correct et optimisé avec

- d'un côté un langage intermédiaire non typé (`Clambda`)
- de l'autre un environnement d'exécution typé (CLR)

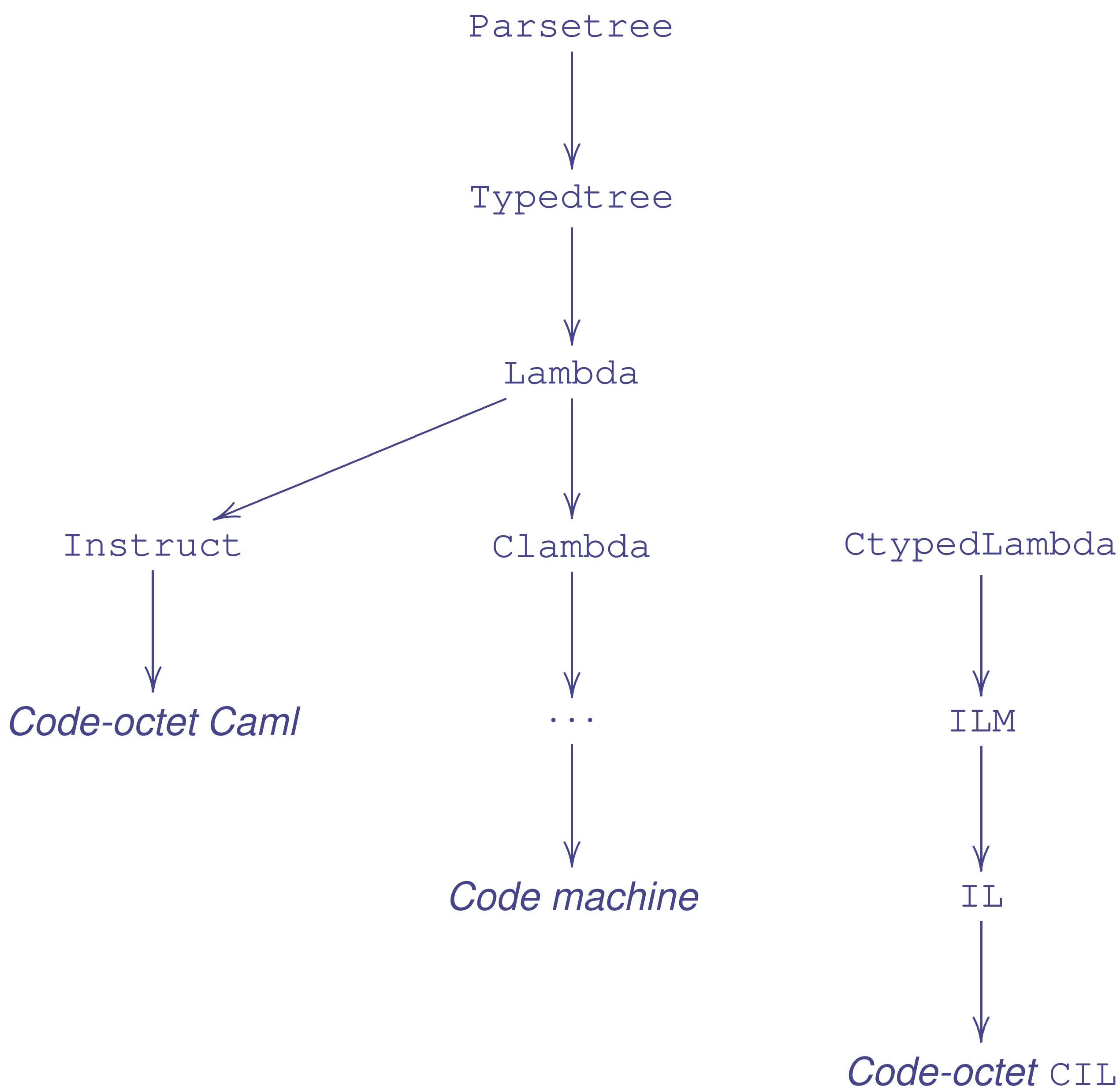
Choix : on tient à la compilation séparée et sur le modèle de Caml, contrairement à SML.NET,

- on ne monomorphise pas les valeurs et les fonctions,
- on a besoin de représentations génériques.

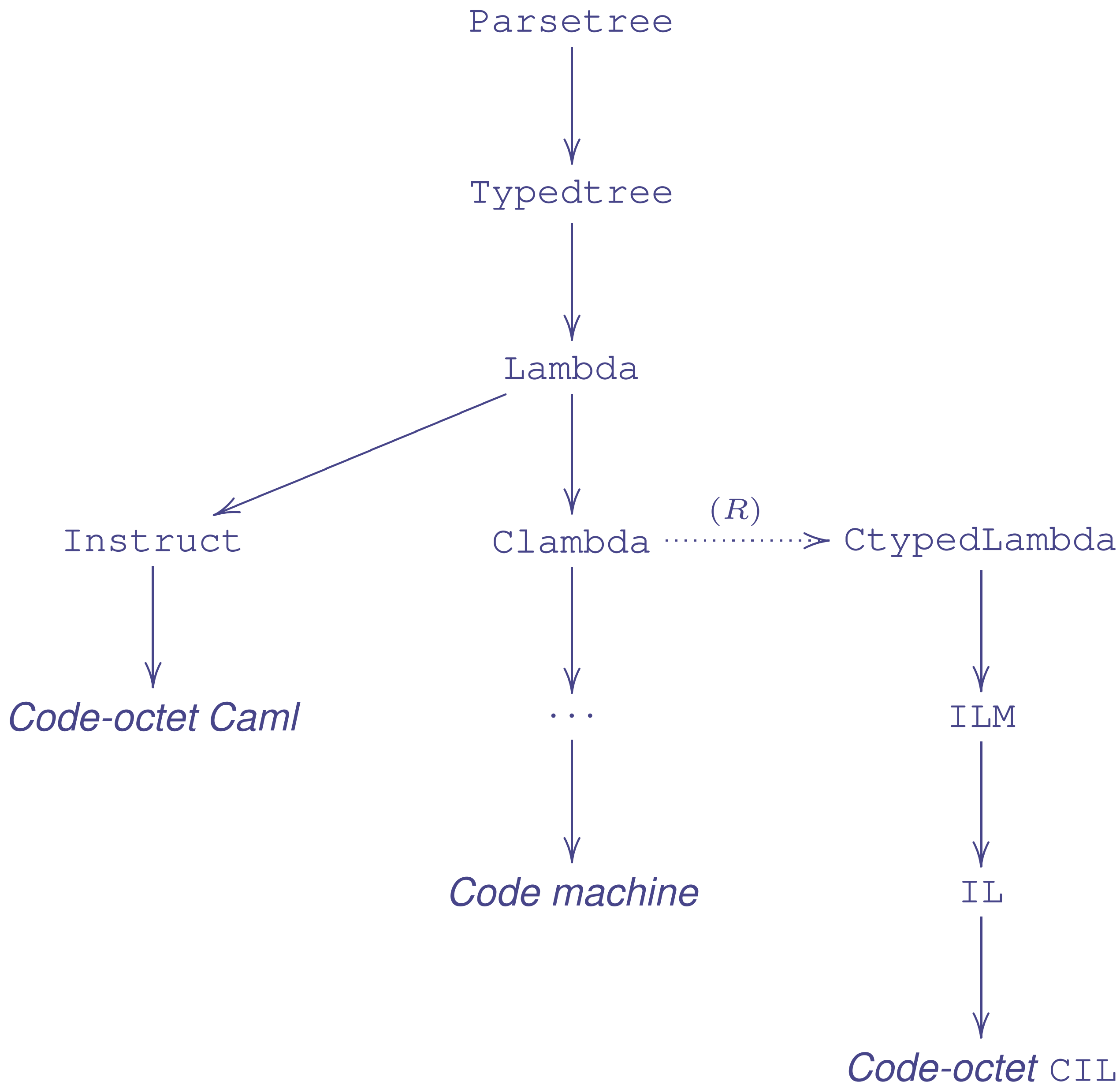
Architecture de OCamlIL



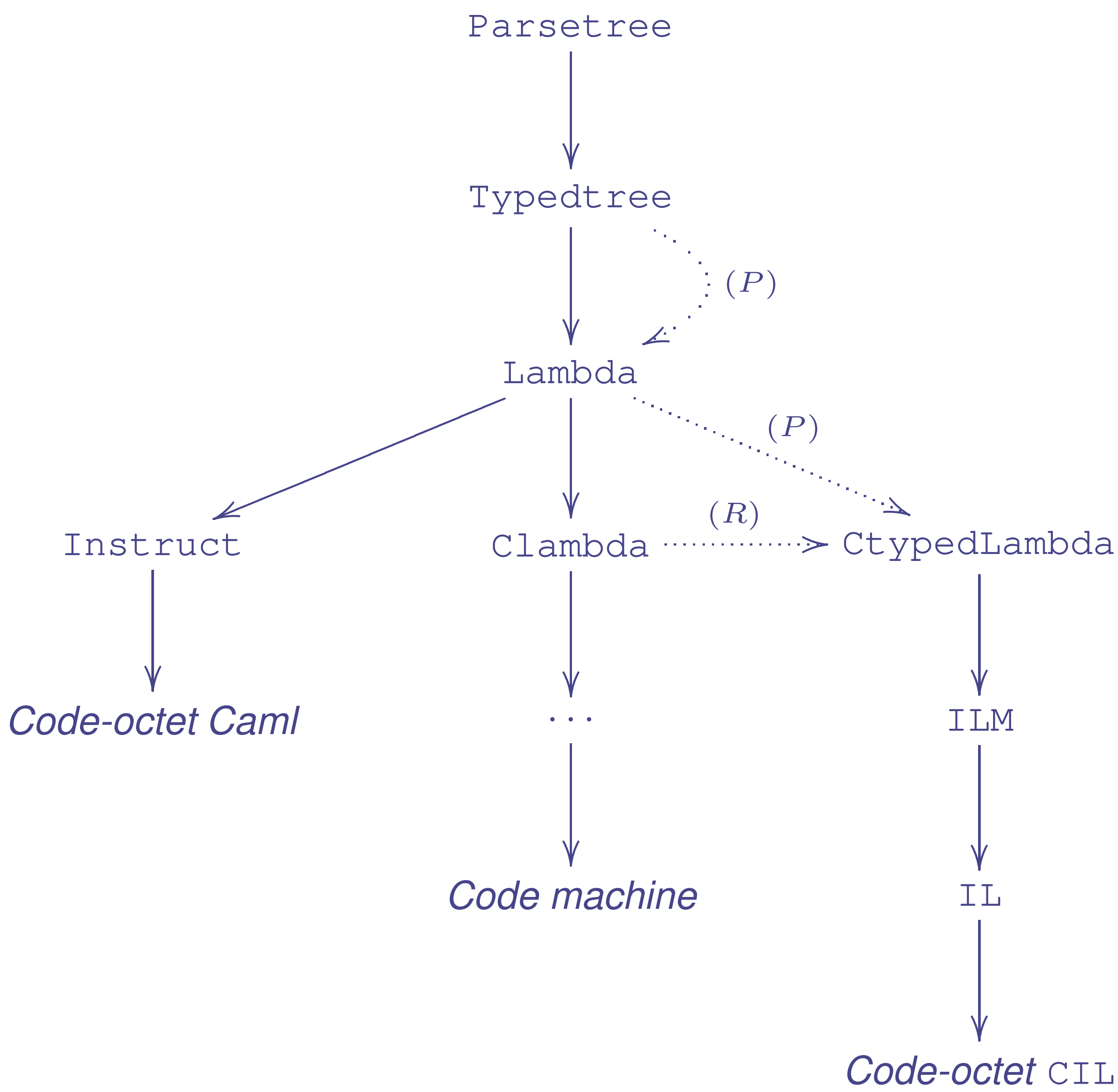
Architecture de OCamIL



Reconstruction de types



Reconstruction et propagation de types



Reconstruction de types

Reconstruction de types

En Caml on a deux types de valeurs à l'exécution, homogènes sur une architecture native :

- les entiers (31 bits avec un tag) : `0`, `1`, `...`
- les pointeurs sur des blocs mémoire : $\rightarrow$ | 3 | $\downarrow$ | 7 | $\downarrow$ |.

On reconstruit les types sur la grammaire :

```
T ::= int | int32 | int64 | nint | unit | void
    | exception | string | float | closure | block
    | anytype
```

- On se sert des primitives pour inférer les types des valeurs.
- On peut discriminer certains types de blocs.

Reconstruction : représentation des valeurs

Types de base :

Caml	int	native int	int32	int64	bool	float	unit	char	string
CTS	int32	IntPtr	int32	int64	int32	float64	null void	int32	char[]

Types anonymes :

Caml	tableaux	n-uplets	variants polymorphes	->
CTS	object[]	object[]	object[]	CamIL.Closure

Types définis :

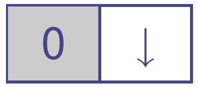
Caml	enregistrements	variants	exceptions	classes
CTS	object[]	object[]	CamIL.Exception	object[]

Types abstraits et polymorphes : object

Reconstruction de types et les variants

Problème : entiers et blocs dans la représentation des types variants.

`type t = Zero | One | Node of t`

Représentations : 0, 1 et 

code O'Caml	code `Clambda`

`let problem a b =`
`  match a with`
`    | Zero -> One`
`    | _ -> b`
	$\mathbf{let\ problem\ =\ closure(Lproblem, 2, a, b,}$
	$\quad\mathbf{if\ prim(IsInt, a)\ then}$
	$\qquad\mathbf{(if\ prim(!=_i, a, 0)\ then\ b\ else\ 1)}$
	$\quad\mathbf{else\ b)}$

type : `t -> t -> t`	type inféré : anytype $\longrightarrow$ int $\longrightarrow$ int

$\Rightarrow$ erreur de compilation sur `problem One (Node Zero)`

Reconstruction de types et les variants

- Solution 1 = un algorithme de retypage moins agressif, avec pour conséquences :
  - des types trop génériques (même quand les variants ne sont pas concernés),
  - trop d'opérations de (dés)-encapsulation altérant les performances.
- Solution 2 (retenue) = modifier l'implantation des variants en représentant tous les constructeurs (même constants) par des blocs :
  - la représentation `Clambda` des variants devient homogène,
  - c'est une modification précoce du compilateur O'Caml.

Reconstruction de types et les variants

Avec ce changement :

```
type t = Zero | One | Node of t
```

Représentations : $\boxed{0}$, $\boxed{1}$ et $\boxed{2 \mid \downarrow}$

code O'Caml	code `Clambda`

`let problem a b =`
`  match a with`
`    | Zero -> One`
`    | _ -> b`
	$\mathbf{let\ problem} = \mathbf{closure}(\mathbf{Lproblem}, \mathbf{2}, \mathbf{a}, \mathbf{b},$
	$\quad \mathbf{switch\ a}$:
	$\qquad \mid \mathbf{0} \to \mathbf{makeblock}(\mathbf{1})()$
	$\qquad \mid \_ \to \mathbf{b})$

type : `t -> t -> t`	type inféré : block $\to$ block $\to$ block

Propagation de types

Propagation de types

L'information de types est disponible sur l'arbre de syntaxe, mais il y a de nombreuses manipulations de code en aval (principalement l'algorithme de compilation du filtrage de motifs).

```
type t = A of int * t | B of int

let f = function
    A(0,u) , B i -> A(i,u)
  | ( A(x,_) , A(y,_) | B x , B y ) -> B(x+y)
  | A(_,u) , _ -> u
  | _ , _ -> B 0
```

Propagation de types : `Clambda`

```
param ->
  catch 1 x' y'
    let left = (field 0 param) in
       switch left
          0 -> let x = (field 0 left) in
                  catch 3
                    if (x != 0) then <jump 3> else
                       let right0 = (field 1 param) in
                          switch right0
                           0 -> <jump 3>
                            1 -> [makeblock 0 (field 0 right0)
                                                (field 1 left)]
                  c-with(3) let right1 = (field 1 param) in
                               switch right1
                                 0 -> <jump 1 x (field 0 right1)>
                                 1 -> (field 1 left)
          1 -> let right = (field 1 param) in
                  switch right
                    0 -> [1: 0]
                    1 -> <jump 1 (field 0 left) (field 0 right)>
  c-with(1) [makeblock 1 (x'+y')]
```

Propagation de types : Ctypedlambda

```
param:t*t ->
  catch 1 (x':int) (y':int)
    let left:t = (field 0 param) in
       switch left
          0 -> let x:int = (field 0 left) in
                  catch 3
                    if (x != (0:int)):bool then <jump 3>:t else
                       let right0:t = (field 1 param) in
                          switch right0
                            0 -> <jump 3>:t
                            1 -> [makeblock 0 (field 0 right0):int
                                                  (field 1 left):t]:t
                  c-with(3) let right1:t = (field 1 param) in
                               switch right1
                                 0 -> <jump 1 x (field 0 right1):int>:t
                                 1 -> (field 1 left):t
          1 -> let right:t = (field 1 param) in
                  switch right
                     0 -> [1: 0]:t
                     1 -> <jump 1 (field 0 left):int (field 0 right):int>:t
  c-with(1) [makeblock 1 (x'+y'):int]:t
```

Propagation de types : Ctypedlambda

```
param:t*t ->
  catch 1 (x':int) (y':int)
    let left:t = (field 0 param) in
       switch left
          0 -> let x:int = (field 0 left:t.A) in
                  catch 3
                    if (x != (0:int)):bool then <jump 3>:t else
                       let right0:t = (field 1 param) in
                          switch right0
                            0 -> <jump 3>:t
                            1 -> [makeblock 0 (field 0 right0:t.B):int
                                              (field 1 left:t.A):t]:t
                  c-with(3) let right1:t = (field 1 param) in
                               switch right1
                                 0 -> <jump 1 x (field 0 right1:t.A):int>:t
                                 1 -> (field 1 left:t.A):t
          1 -> let right:t = (field 1 param) in
                  switch right
                     0 -> [1: 0]:t
                     1 -> <jump 1 (field 0 left:t.A):int (field 0 right:t.B):int>:t
  c-with(1) [makeblock 1 (x'+y'):int]:t
```

Propagation de types : Ctypedlambda

```
param:t*t ->
  catch 1 (x':int) (y':int)
    let left:t = (field 0 param) in
       switch left
          0 -> let x:int = (fldtag(0) 0 left:t) in
                  catch 3
                    if (x != (0:int)):bool then <jump 3>:t else
                       let right0:t = (field 1 param) in
                          switch right0
                            0 -> <jump 3>:t
                            1 -> [makeblock 0 (fldtag(1) 0 right0:t):int
                                              (fldtag(0) 1 left:t):t]:t
                  c-with(3) let right1:t = (field 1 param) in
                               switch right1
                                 0 -> <jump 1 x (fldtag(0) 0 right1:t):int>:t
                                 1 -> (fldtag(0) 1 left:t):t
          1 -> let right:t = (field 1 param) in
                  switch right
                     0 -> [1: 0]:t
                     1 -> <jump 1 (fldtag(0) 0 left:t):int (fldtag(1) 0 right:t):int>:t
  c-with(1) [makeblock 1 (x'+y'):int]:t
```

Propagation : représentation des valeurs

Types de base :

Caml	int	native int	int32	int64	bool	float	unit	char	string
CTS	int32	IntPtr	int32	int64	**bool**	float64	null void	**char**	char[]

Types anonymes :

Caml	tableaux	n-uplets	variants polymorphes	->
CTS	object[]	object[]	object[]	CamlL.Closure

Types définis :

Caml	enregistrements	variants	exceptions	classes
CTS	**classes dédiées**	**classes dédiées**	CamIL.Exception	object[]

Types abstraits et polymorphes : object

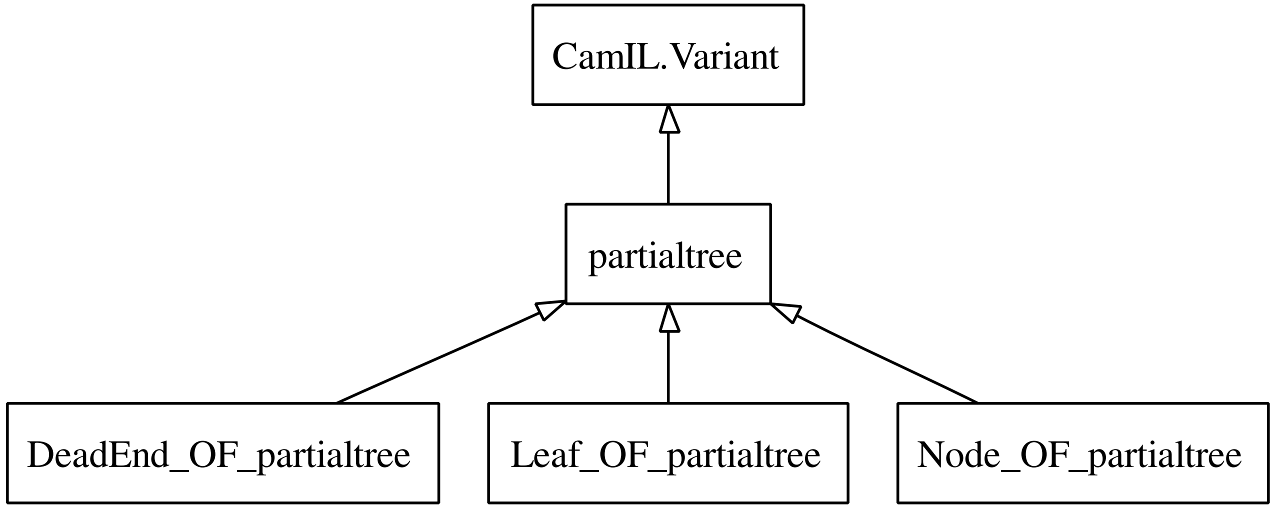
Représentation des enregistrements

<table>
<tr><th>Type Caml</th><th>Type CTS</th></tr>
<tr><td><code>type people = {name:string; age:int}</code></td><td><code>class people extends CamIL.Record {</code>
    <code>char[] name;</code>
    <code>int32 age;</code>
    implantations méthodes virtuelles
<code>}</code></td></tr>
<tr><td><code>type ’a label = {lbl:string; v:’a}</code></td><td><code>class label extends CamIL.Record {</code>
    <code>char[] lbl;</code>
    <code>object v;</code>
    implantations méthodes virtuelles
<code>}</code></td></tr>
</table>

- `int32 compare(CamIL.Record)` et `bool equals(CamIL.Record)`,
- `CamIL.Record duplicate()`,
- `int32 hashcode()`.

Représentation des variants

```
type 'a partialtree =
      | DeadEnd
      | Leaf of 'a
      | Node of 'a partialtree * 'a partialtree
```



- `int32 tag`
- `compare`, `equals` et `hashcode`

Propagation de types

Par rapport à la reconstruction, la propagation des types :

- permet d'attribuer des types plus précis aux primitives, et de choisir des représentations plus adaptées
  - meilleure lisibilité (interopération, débogage)
  - meilleure performance du code produit (jusqu'à $\times 2$)
- modifications plus profondes du compilateur Caml
- pose problème pour `Obj.magic`

Représentation des fermetures et des exceptions

Représentation des fermetures

Enjeux :

- choisir une représentation des fermetures,
- étendre le CLR par un mécanisme d'application général.

Possibilités d'implantation :

- une classe par fermeture
- un type délégué associé à la fermeture
- une classe unique avec un dispatch géré par du code engendré

Représentation des fermetures dans OCamIL

Principes :

- une classe par fermeture syntaxique,
- un environnement typé précisément (implanté par des champs de la classe),
- une méthode statique `exec` typée exactement pour l'application totale,
- une méthode d'instance `object apply(object)` pour l'application générique.

Représentation des exceptions

Les exceptions OCamIL :

- sont représentées par `CamIL.Exception` qui hérite de `System.Exception`
- implantent le bloc Caml de l'exception de manière générique (avec `object[]`)

Conséquences :

- bon pour l'interopération
  - on peut capturer des exceptions CLR quelconques en OCamIL
  - on peut lancer ou traiter des exceptions OCamIL en C#
- peu efficace sur .NET

Interopérabilité

Interopérabilité : bas niveau

Repose sur une interface externe de bas-niveau avec des méthodes statiques .NET.

```
external il_getenv: string -> string =
"string" "System.Environment::GetEnvironmentVariable" "string"
;;

let getenv var = match il_getenv var with
    | "" -> raise Not_found
    | s -> s
;;
```

Exemple : implantation de `Sys.getenv`

Interopérabilité : IDL

Trait	CTS	O'Caml	Trait	CTS	O'Caml
classes	$\surd$	$\surd$	sous-typage	$\surd$	$\surd$
liaison tardive	$\surd$	$\surd$	héritage $\equiv$ sous-typage ?	oui	non
liaison statique	$\surd$	$\surd^*$	surcharge	$\surd$	
typage statique	$\surd$	$\surd$	héritage multiple	*	$\surd$
typage dynamique	$\surd$		classes paramétrées	$\surd^*$	$\surd$

- **IDL** = une intersection des deux modèles : confondant héritage et sous-typage, sans surcharge ou méthode binaire. Pas d'héritage multiple ni de classes paramétrées.
- **O'Jacaré.Net** : compilateur de l'IDL (orienté vers O'Caml).

O'Jacaré.Net : classe Point

Fichier `point.idl`

```
class Point {
  int x;

  [name default_point] <init> ();
  [name point] <init> (int);

  void moveTo(int);
  string toString();
  boolean eq(Point);
}
```

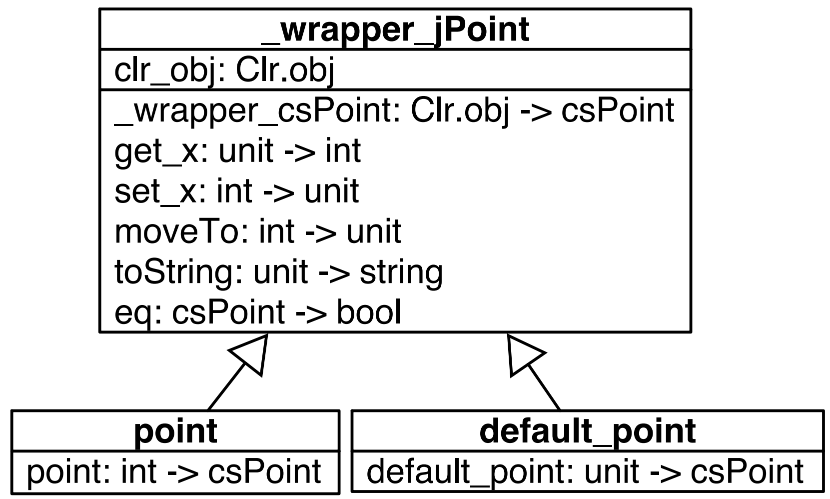
Génère : `point.ml`

Type objet `csPoint`

Wrapper `_wrapper_csPoint`

Classes utilisateur

`default_point, point`



O'Jacaré.Net : classe ColoredPoint

Fichier `point.idl`

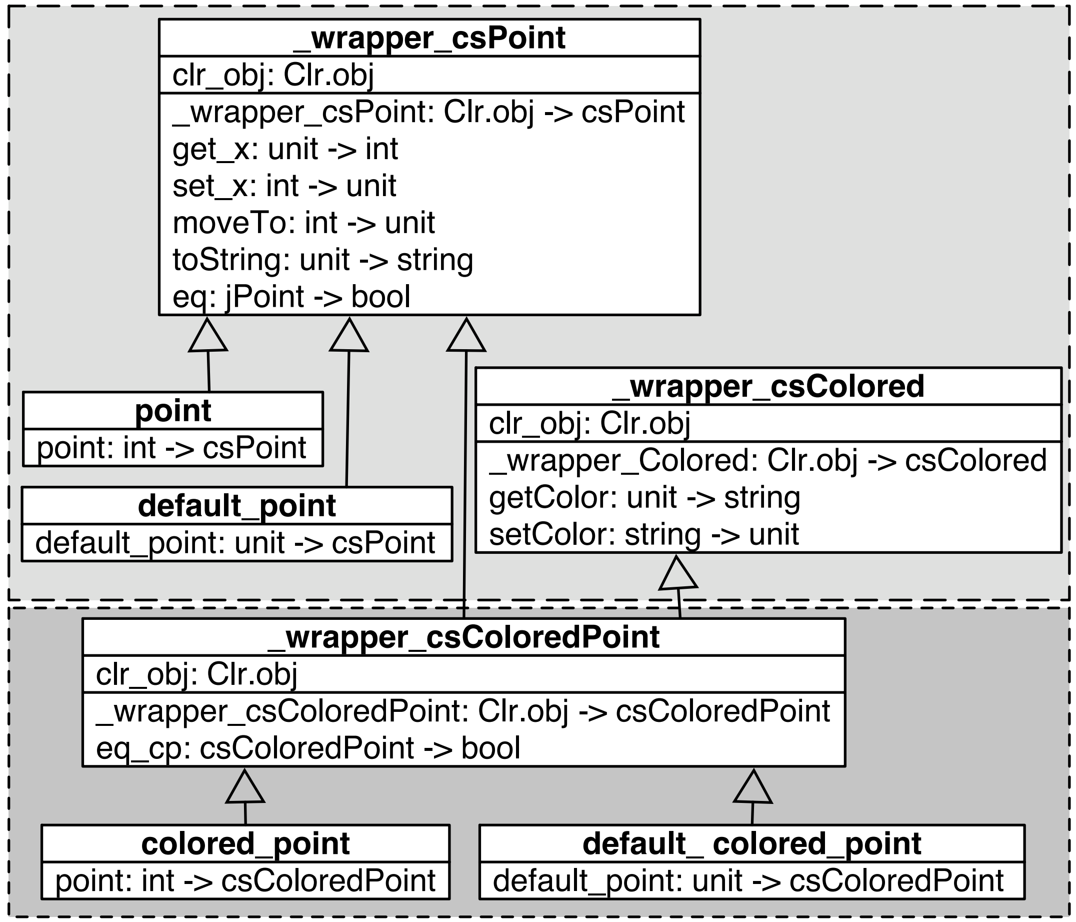
```
class ColoredPoint extends Point
  implements Colored {

    [name default_colored_point]
      <init> ();

    [name colored_point]
      <init> (int,string);

    [name eq_cp]
      boolean eq(ColoredPoint);
}
```

Génère : `point.ml`



O'Jacaré.Net: remarques finales

- Fondé sur l'API `Reflection`.
- Orienté O'Caml mais un mécanisme de callback permet une communication dans les deux sens.
- Vérification dynamique de la cohérence de l'IDL.
- Combinaison des deux modèles de classes :
  - héritage multiple en O'Caml de classes C#,
  - downcast d'objets C# en O'Caml.
- Autres constructions interfacées :
  - types valeurs structurés, interfacés via leur type référence associé,
  - énumérations, interfacées par des types sommes.

Réalisations et Applications

Maturité

Portage des librairies :

- Librairie standard
- Graphics
- Threads
- Dynlink

Compatibilité :

- Compilateur bootstrappé
- Toplevel

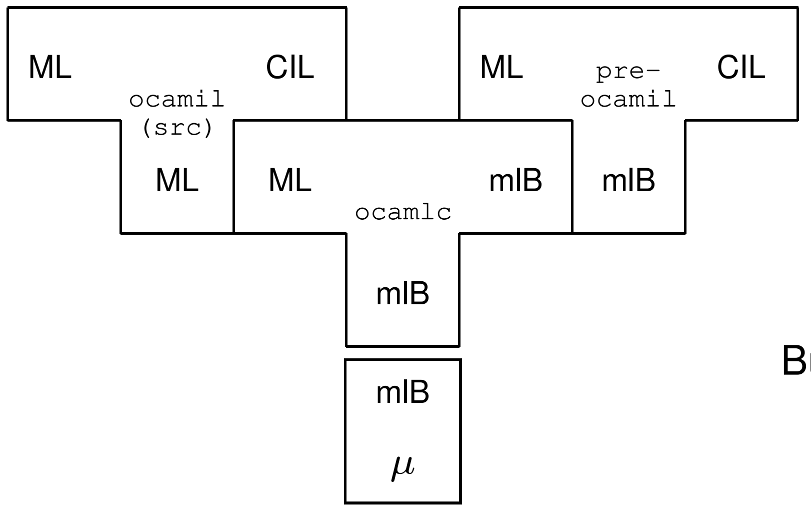
Utilisation de O'Jacaré.Net pour interfacer l'API `Reflection`.

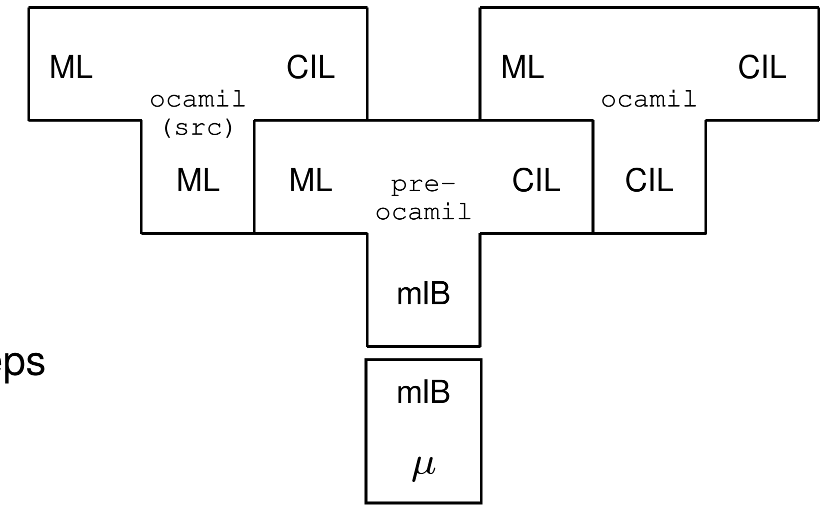
Performances

	ocamlopt	ocamlc	ocamil	fsc	smlnet
Fft	0,12	1,00	0,46	0,52	
KB	0,15	1,00	17,83	13,85	9,52
Boyer	0,21	1,00	11,10	10,41	7,67
Nucleic	0,19	1,00	0,76	0,49	
Quicksort	0,03	1,00	0,20	0,43	
SoliLet	0,12	1,00	0,88	0,43	

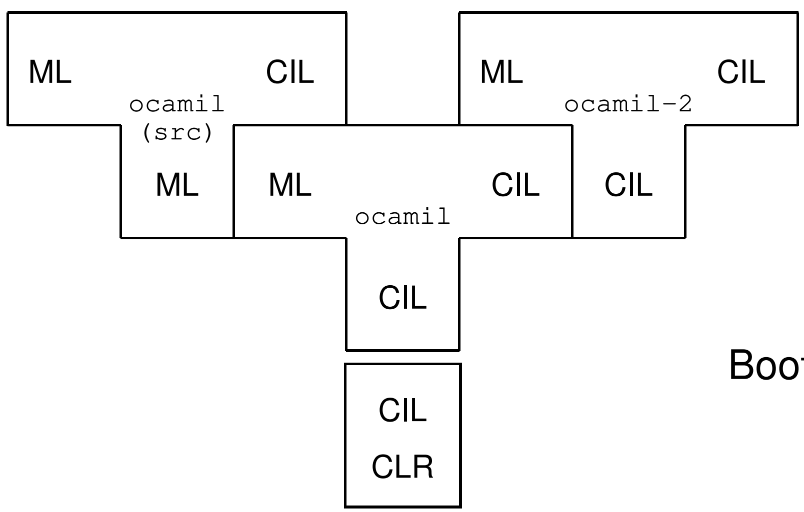
- cas polymorphe, très fonctionnel, exceptions : code-octet Caml bien meilleur que .NET
- cas monomorphe, peu fonctionnel ou impératif : on se situe entre le code-octet et le code natif Caml
- F# $\simeq$ OCamIL, SML.NET un peu meilleur (monomorphisation globale)

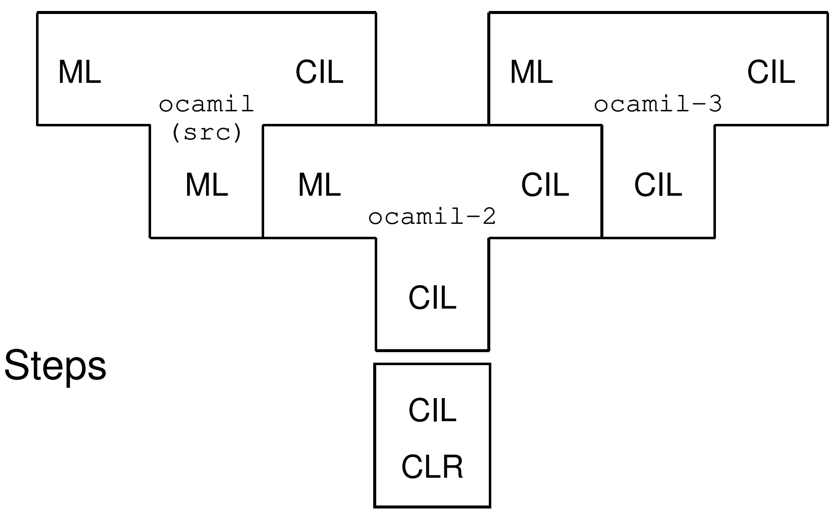
Construction et bootstrap





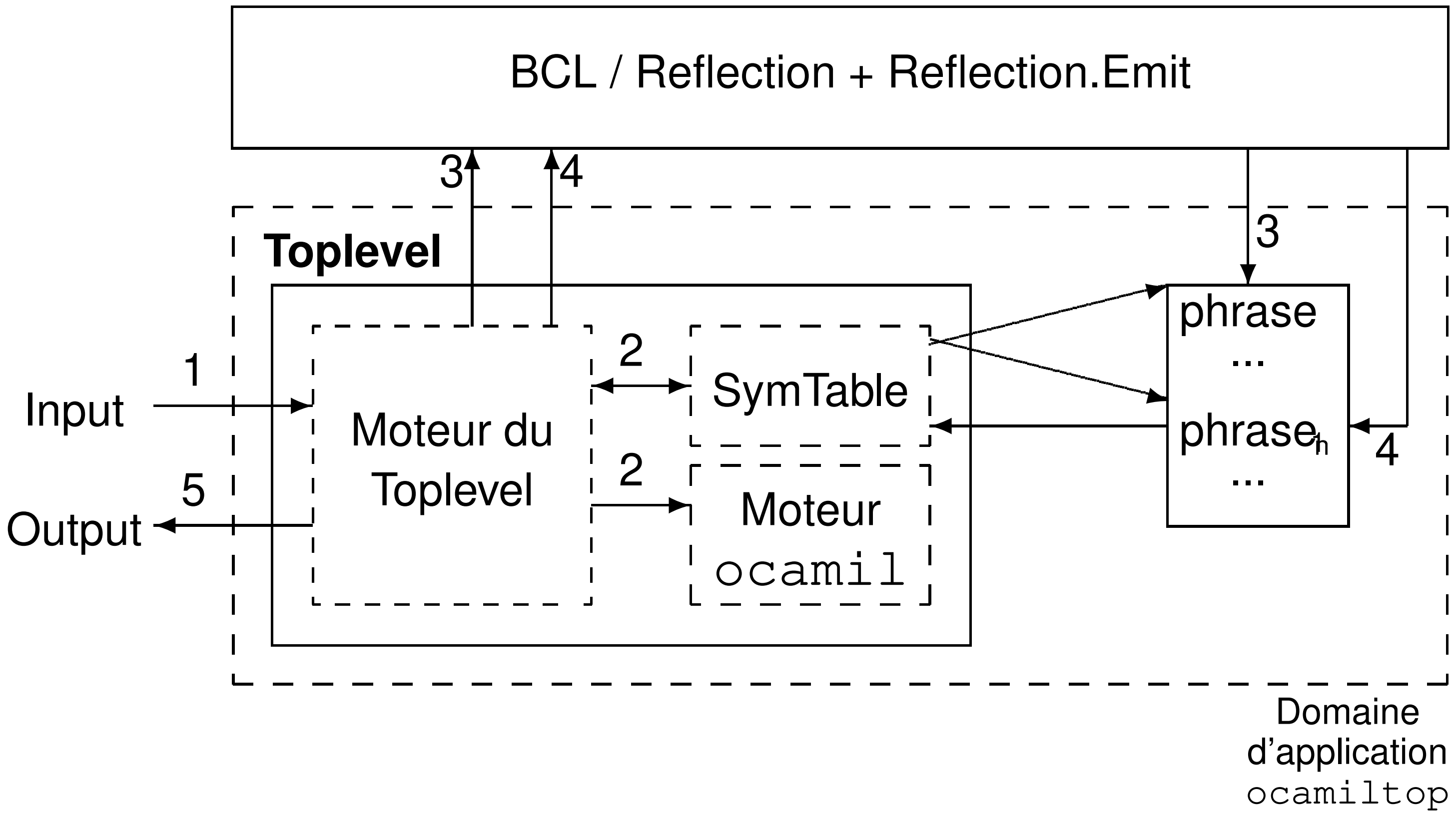
Building Steps





Bootstrapping Steps

Toplevel



BCL / Reflection + Reflection.Emit
3
4
3
Toplevel
1
Input
5
Output
Moteur du Toplevel
2
SymTable
2
Moteur ocamil
phrase
...
phrase$_n$
...
4
Domaine d'application ocamiltop

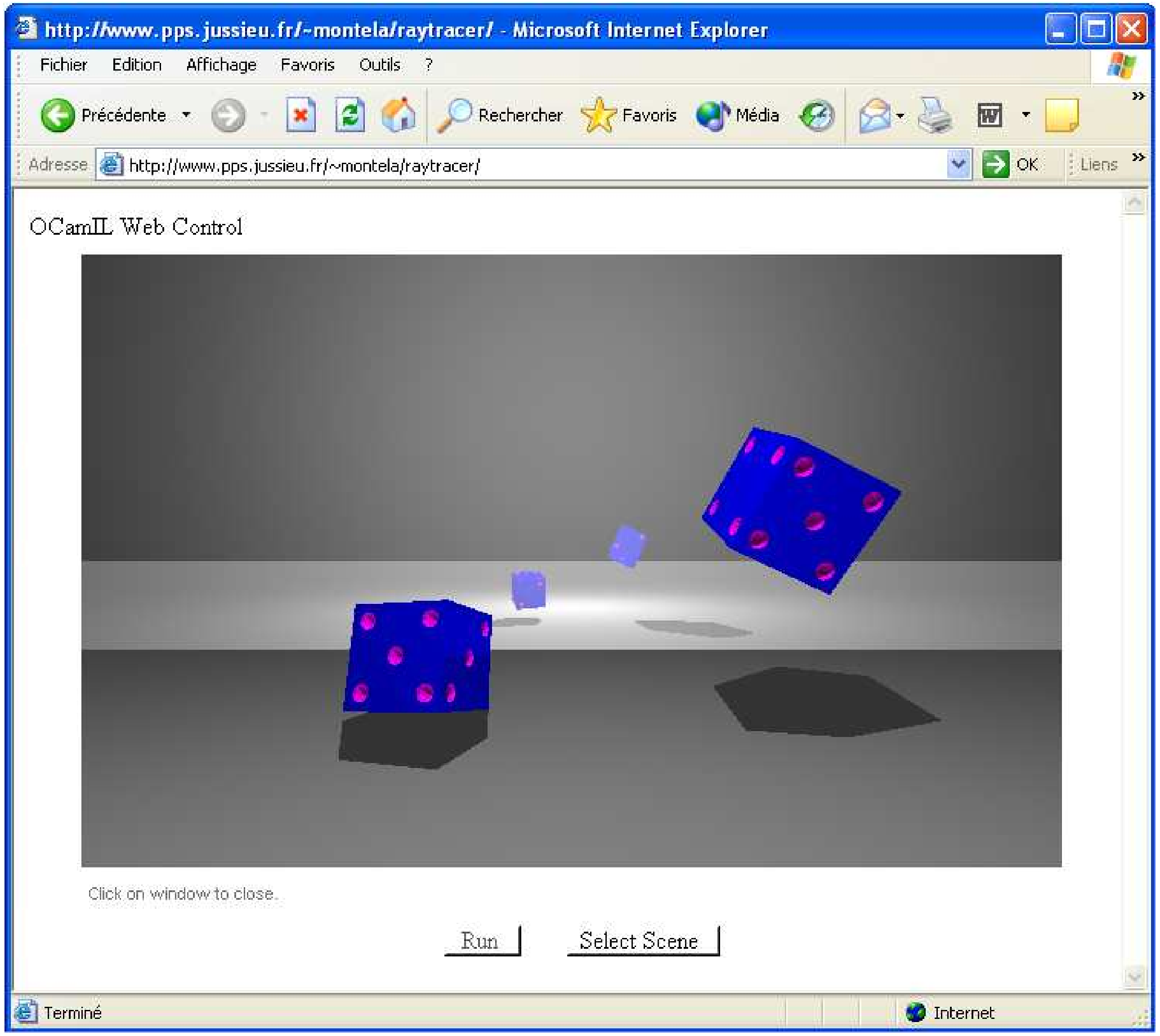
Sérialisation sûre : principes

- dans l'univers géré, les types sont sérialisés et la désérialisation est toujours sûre
  - elle n'échoue jamais (type de retour `System.object`)
  - les problèmes surviennent au downcast (levée d'une exception `InvalidCastException`)
- on l'exploite directement dans OCamIL (types de base, types anonymes, types définis, fermetures, objets)

Cependant :

- les erreurs de transtypage portent sur les types CTS dynamiques, pas directement sur les types Caml statiques
- les erreurs de transtypage peuvent apparaître bien après la désérialisation

Appliquette



Conclusion et travaux futurs

Réalisations

- un compilateur complet
  - tous les traits de O'Caml
  - performances semblables à F#
  - bootstrap + toplevel
- l'investigation de deux solutions antagonistes au problème des informations de types
- l'interopérabilité via le compilateur d'IDL O'Jacaré.Net
- une sérialisation sûre
- un bilan sur la distance qui sépare les langages fonctionnels des machines virtuelles modernes

Travaux futurs

- Sur le compilateur :
  - optimisations (entre autres via des analyses statiques),
  - exploitation des Generics,
  - enrichissement de l'IDL (délégués, Generics),
  - intégration avec le monde non-géré,
  - amélioration de la sérialisation sûre,
  - back-end Java.
- Sur les outils :
  - debugging (propager les informations du source),
  - IDE (Visual Studio, Eclipse) : debugging interactif avec exploration des valeurs, typage à la saisie . . .

O'Caml et machines virtuelles génériques

Pour un langage avec typage statique, il est intéressant de conserver les types pour :

- l'efficacité sur les plates-formes modernes,
- l'ouverture sur l'extérieur,
- des applications comme la sérialisation sûre.

Cependant, il est difficile de profiter du typage statique pour les performances sur ces plates-formes.

La prochaine machine virtuelle généraliste ?

Portable comme Java et multi-langages comme .NET !

On a déjà :

- GC commun pour l'interopération,
- threads communs,
- du polymorphisme (Generics),
- implantation de la récursion terminale (tailcall/jmp).

On aimerait aussi :

- une représentation optimisée pour les fermetures,
- des exceptions moins lourdes,
- la prise en charge de types structurés anonymes.

Merci !