OCamIL, un compilateur Objective Caml vers .NET

Emmanuel Chailloux* Raphaël Montelatici† Bruno Pagano‡

***Abstract*— We present a first version of our Objective Caml compiler, called OCamIL, for .NET. Our goal is to understand whether this new generation of virtual machines and runtime environment can help us compile ML programs and produce executables of reasonable efficiency. Our main constraint is to be compatible with the original language, and its advanced programming features (functional values, exceptions, parameterized modules, objects).**

***Index Terms*— Compilation, functional languages, static type-checking, virtual machine, .NET, Objective Caml.**

I. Introduction

La plate-forme .NET[1] se veut un creuset où s'intègrent différents langages dans un cadre commun tant du point de vue du système de types CTS (*Common Type System*) que de l'environnement d'exécution CLR (*Common Language Runtime*). Chaque compilateur de langage produit un code intermédiaire portable MSIL (*MicroSoft Intermediate Language*) qui est ensuite assemblé et exécuté. Une application .NET ne s'intéresse pas aux langages sources des composants qu'elle manipule. Leur conformité au système de types garantit leur interopérabilité.

La liste des langages portés sur cette plate-forme est conséquente. On peut citer de grands classiques comme C#, J# ou A#[2] mais aussi des langages moins connus comme Eiffel[3], Scheme[4], Sml[5], F#[6], Mercury[7].

Bien que l'implantation principale .NET fonctionne uniquement sous Windows, des initiatives «Open Source» voient le jour comme Rotor[8] pour Windows et Unix BSD, et Mono[9] pour Linux. On rejoint alors la problématique de Java «COMPILE ONCE, RUN EVERYWHERE».

* Equipe Preuves, Programmes et Systèmes (UMR 7126),
Université Pierre et Marie Curie, 175 rue du Chevaleret, 75013 Paris, France.
Email : Emmanuel.Chailloux@pps.jussieu.fr

† Equipe Preuves, Programmes et Systèmes (UMR 7126),
Université Denis Diderot, 175 rue du Chevaleret, 75013 Paris, France.
Email : Raphael.Montelatici@pps.jussieu.fr

‡ Esterel technologies R&D,
679 Av Julien Lefèbvre , 06270, Villeneuve-Loubet, France.
Email : Bruno.Pagano@esterel-technologies.com

[1] `www.microsoft.com/net`
[2] `www.usafa.af.mil/dfcs/bios/mcc_html/a_sharp.html`
[3] `www.msdnaa.net/Resources/display.aspx?ResID=811`
[4] `www-sop.inria.fr/mimosa/fp/Bigloo`
[5] `www.cl.cam.ac.uk/Research/TSG/SMLNET/`
[6] `research.microsoft.com/projects/ilx/fsharp.aspx`
[7] `www.cs.mu.oz.au/research/mercury/dotnet.html`
[8] `msdn.microsoft.com/net/sscli`
[9] `www.go-mono.com/`

Au final on espère avoir une plate-forme multi-langages à bibliothèque d'exécution unique, sûre et efficace, qui pourrait être multi-systèmes.

Pour apporter notre pierre à l'édifice et pour vérifier les propriétés annoncées on se propose de réaliser un compilateur .NET pour notre langage de prédilection, Objective Caml[1]. Objective Caml est un dialecte de ML, une grande famille de langages. C'est un langage fonctionnel/impératif, statiquement typé, polymorphe paramétrique, muni d'un mécanisme d'exception, possédant une extension objet et un système de modules paramétrés. Son pouvoir d'expression est important[2],[3], sa sémantique de par sa tradition est formalisée et de plus son implantation, y compris le compilateur de code-octet (*byte-code*), produit des programmes efficaces.

L'amélioration des techniques de compilation des langages fonctionnels sur les vingt dernières années permet de produire du code efficace pour des processeurs réels ou des machines virtuelles spécialisées. Ces techniques ne sont pas toujours adaptées aux nouvelles machines virtuelles pour langages objets comme la JVM (*Java Virtual Machine*) ou le CLR de .NET. En effet pour garantir la sûreté du chargement dynamique de code-octet, l'environnement d'exécution de ces machines effectue des vérifications de types sur le code à charger. L'adage «Runtime Tags Aren't Necessary» de [4] n'est plus de mise.

Le compilateur OCamIL[10] se veut un vecteur de diffusion du langage Objective Caml. Sa principale contrainte est alors la compatibilité avec Objective Caml. Pour cela il dérive une nouvelle branche de production de code au sein même du compilateur Objective Caml. Il rencontre alors les difficultés de génération du code IL typé dans la mesure où Objective Caml, une fois un programme typé, n'utilise plus cette information dans le code produit. Il devient donc nécessaire de reconstruire les informations de type au niveau d'un des langages intermédiaires d'Objective Caml, sachant que le programme d'origine était typé. D'autres difficultés, plus classiques, apparaissent pour la compilation de la partie fonctionnelle du langage vers une machine virtuelle spécialisée pour les langages objets. Les portages de la bibliothèque standard (`stdlib`) et de la bibliothèque `graphics` ont montré l'interopérabilité avec le langage IL et les bibliothèques existantes. Comme le compilateur OCamIL est principalement écrit en Objective Caml, un test important de compatibilité a été réalisé par son auto-amorçage (*bootstrap*). Ainsi OCamIL peut être distribué sous forme d'une application .NET.

On présente tout d'abord les caractéristiques principales de la plate-forme .NET pour la compilation d'un «nouveau» lan-

[10] `www.pps.jussieu.fr/~montela/ocamil`

gage pour ensuite décrire la démarche suivie pour la réalisation de OCamIL. Bien que la compatibilité avec Objective Caml ait été notre critère principal, nous avons tenté de conserver une efficacité raisonnable pour le code produit. Pour le vérifier nous comparons OCamIL avec d'une part son illustre ancêtre Objective Caml et d'autre part avec le compilateur F# (qui traite un sous-ensemble du langage Objective Caml) et le compilateur SML.NET pour un autre dialecte ML. D'autres travaux sur la compilation de langages fonctionnels vers la JVM ou .NET sont commentés ainsi que des travaux sur le typage de code intermédiaire. Notre conclusion détaille les orientations de nos futurs travaux.

II. PLATE-FORME .NET

La plate-forme .NET est présentée comme la prochaine technologie de référence dans le développement d'applications Windows, autant pour les applications de bureautique que de serveurs Web ou encore toutes sortes de composants logiciels. Elle est censée apporter de nouvelles garanties en matière de gestion de sécurité et d'erreurs, et permettrait par exemple de remédier aux travers de Windows en matière de bibliothèques partagées[5] [11].
On s'intéresse ici uniquement à cette plate-forme à travers l'œil du développeur, et plus spécifiquement du développeur d'un compilateur. La plate-forme propose un environnement d'exécution CLR essentiellement composée d'une machine virtuelle à pile[12] et d'une bibliothèque de support BCL[13]. La machine virtuelle vérifie que du code-octet MSIL respecte le modèle de classes typées CTS. Ce code-octet est ensuite expansé vers du code exécutable par l'environnement d'exécution comme présenté à la figure 1. Un paquetage PE (*Portable Executable*) contient des instructions MSIL avec d'éventuels fichiers de ressources.

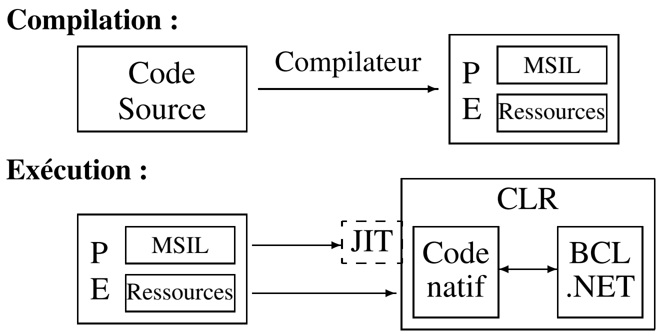

Fig. 1. Compilation et exécution

On détaille ci-dessous les principales caractéristiques de la plate-forme :

1) Le code-octet MSIL: C'est un assembleur typé : tout emplacement où des valeurs sont stockées, passées ou utilisées est muni d'un type. On distingue les types-références *Reference Types* (pour des objets alloués sur le tas) et les types-valeurs *Value Types* (alloués sur la pile uniquement) qui peuvent être structurés. Des instructions `box` et `unbox` permettent de jongler entre les deux représentations. Les objets comportent des champs et des méthodes, celles-ci disposant de variables locales en plus de la pile. En ce qui concerne le contrôle, la machine permet les appels de méthodes (éventuellement avec liaison tardive, ou même via une indirection), les branchements au sein d'une même méthode et gère les exceptions.

2) Déploiement: Le composant de base en .NET est l'assemblage (*assembly*). Un assemblage peut être signé à l'aide d'une clé cryptographique (*strong naming*) : la machine hôte peut alors accorder sa confiance au code qu'il contient, ce qui permet de partager ce code, en plaçant l'assemblage dans le cache global des assemblages[14], un répertoire spécial d'installation. Ce mécanisme est également exploité afin de faciliter la gestion de versions et de régionalisations distinctes de composants logiciels.

3) Sûreté à l'exécution: Parce qu'un exécutable peut provenir de différents compilateurs, le code MSIL est typé statiquement lors de son assemblage pour éviter les incohérences de typage de ces différentes sources. De plus, tout assemblage produit à destination du CLR peut être soumis à un outil de vérification `PEVerify` qui détecte des incohérences de gestions de piles, des erreurs dans la résolution des références à des assemblages externes (appels erronés à des fonctions externes par exemple), et même certaines erreurs de types pouvant survenir à l'exécution. L'information de typage est conservée à l'exécution, et en cas d'erreur dynamique de types, le CLR déclenche une exception. Ces caractéristiques sont très appréciables pour la mise au point d'un compilateur ciblant MSIL. Le code MSIL soumis aux contraintes de types et de vérification est appelé «code géré» (*managed code*). La plate-forme permet toutefois des appels à des portions de code non géré, ce qui est inévitable pour certaines opérations de bas niveau.

On note de plus que l'environnement est muni d'un mécanisme de récupération automatique de mémoire (*GC*[15]) de type *Mark and Compact*[5], ce qui affranchit le développeur des soucis de gestion de la mémoire.

4) Performances: La plate-forme met en œuvre un mécanisme de compilation *Just In Time* systématique (chaque méthode est compilée lors de son premier appel). Il est également possible de court-circuiter ce mode de fonctionnement en précompilant un assemblage en une image native.
D'autre part, l'appel de méthode peut se faire de façon récursive terminale, ce qui s'avère particulièrement utile pour l'implantation des langages fonctionnels.

5) Réflexion: Enfin, la plate-forme propose une bibliothèque d'auto-inspection (*reflection*) qui autorise la gestion de code dynamique (génération, chargement et exécution).

III. COMPILATEUR OCAMIL

A. *Schéma général*

Notre objectif principal est de réaliser un portage de Objective Caml vers la plate-forme .NET totalement compatible avec l'implantation de référence, les soucis d'efficacité étant jugés secondaires dans un premier temps.

[11] «The End of DLL Hell», `msdn.microsoft.com/netframework/`
[12] VES : Virtual Execution System.
[13] Base Class Library.
[14] GAC : *Global Assembly Cache*
[15] *Garbage Collector* ou Glaneur de Cellule

Adapter intégralement un langage fonctionnel évolué comme Objective Caml, possédant des traits impératifs et objets, avec un système de modules paramétrés et muni d'un système de typage statique avec inférence de types n'est pas chose facile. Notre expérimentation consiste à écrire un compilateur qui se branche de manière assez tardive sur le mécanisme de compilation standard de Caml, après les phases d'analyse grammaticale et d'inférence de types. Nous récupérons donc le programme à compiler déjà traduit vers un langage intermédiaire interne `Clambda` du compilateur Caml traditionnel, comme illustré sur la figure 2. `Clambda` explicite la gestion de l'environnement des fermetures. Nous introduisons une nouvelle représentation intermédiaire typée `Tlambda` : sa nécessité est discutée à la sous-section III-D.

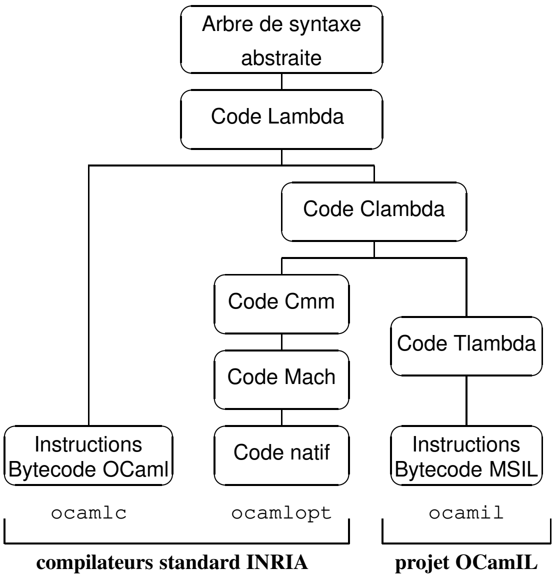

Fig. 2. Branchement de OCamIL sur le compilateur Objective Caml

OCamIL transforme un ensemble de modules Caml en un fichier exécutable portable, comme sur la figure 1. Le code produit délègue un certain nombre de tâches de bas-niveau (gestion de mémoire, entrées-sorties ...) à l'environnement CLR et ses bibliothèques de support.

B. Représentation des données

1) Types de bases: La traduction des types de base Caml en types .NET ne pose pas de problème *a priori*, le système de types de la plate-forme visée est suffisamment riche. Ainsi les types de bases sont transformés de la façon suivante :

Caml	bool	int	float	char	string
.NET	int32	int32	float64	char	StringBuilder

2) Valeurs structurées: Les n-uplets, les tableaux, les enregistrements, les listes et les valeurs de types sommes, sont traditionnellement représentés en mémoire par des blocs dans le tas, caractérisés par leur taille en mots et un octet de *tag* (qui sert par exemple à coder le constructeur d'une valeur d'un type somme).

Ces blocs sont compilés en MSIL comme des tableaux d'objets (`object[]`), ce qui nous oblige souvent à encapsuler les types de bases qui ne sont pas de nature objet.

3) Fermetures: Dans la compilation de Caml, les fermetures modélisent les valeurs fonctionnelles munies de leur environnement, qui sont également des blocs sur le tas. Elles peuvent représenter des fonctions mutuellement récursives (à l'aide de partage et de détection de cycle).

4) Objets Caml: Transposer directement une hiérarchie de classes Objective Caml en une hiérarchie .NET vient immédiatement à l'esprit. Outre les difficultés théoriques que cela pose (en raison des différences dans les modèles objets), cela est rendu difficile par la représentation des objets Caml au sein du code intermédiaire que nous récupérons : ils n'apparaissent plus en tant qu'objets, mais seulement à travers des blocs de champs et de fonctions (le mécanisme de liaison tardive figure explicitement dans le code engendré).

C. Contrôle de l'exécution

1) Application: Dans OCamIL, les fermetures sont implantées comme des classes dérivant d'une classe utilitaire `CamIL.Closure`, possédant des champs destinés à capturer l'environnement de la fermeture et munies de deux méthodes principales : une méthode `exec` qui implante la fonction elle-même lorsqu'elle est appliquée à tous ses arguments, et une méthode `apply: object -> object`, utilisée en cas d'application partielle, qui retourne la nouvelle fermeture obtenue après application du prochain argument attendu.

2) Exceptions: Les exceptions Caml sont directement implantées en tirant partie du mécanisme d'exception de la plateforme cible, au moyen d'une classe `CamIL.Exception` qui hérite de `System.Exception` (la classe mère de toutes les exceptions dans le CTS).

D. (Re)-typage de langage intermédiaire

Le compilateur OCamIL récupère un programme Caml sous forme de code `Clambda`. Ce code n'a plus d'information de type, et pire encore, il utilise certaines optimisations qui exploitent les caractéristiques de l'environnement d'exécution standard. Par exemple l'implantation Caml traite de manière homogène les valeurs entières et les pointeurs sur des blocs en les distinguant par un *bit* de *tag*. Ainsi l'allocation d'un bloc Caml dont une des cases contient un entier ne nécessite pas d'indirection sur cet entier, qui peut être mis en ligne dans la structure du bloc. En revanche de par la représentation des blocs comme tableaux d'objets en MSIL, il est nécessaire d'allouer une encapsulation de l'entier.
Ce genre de manipulation complique la génération de code, et réclame une analyse du code Clambda source, visant à reconstituer une information partielle sur les types mis en jeu. Le tableau suivant montre une génération de code MSIL incorrecte parce qu'elle est ignorante des types (la variable `t` désigne un tableau) :

Source Caml	MSIL	Commentaires
`t.(0) + 1`	`ldloc t`	`t` local sur la pile.
	`ldc.i4.0`	Entier 0 sur la pile.
Code Clambda	`ldelem.ref`	Chargement élément
`(+ (get t 0) 1)`	(*)	(objet) du tableau.
	`ldc.i4.1`	Entier 1 sur la pile.
	`add`	Addition.

Au niveau de (*), le sommet de la pile contient une référence sur un objet alors que l'instruction `add` attend un entier. Nous avons recours à un nouveau langage intermédiaire `Tlambda`, redécoré avec des types et qui comporte les instructions de transtypage adéquates. Dans l'exemple précédent, cela permet d'engendrer une instruction `unbox` au niveau de (*). La garantie de sûreté du typage est en fait héritée des propriétés du typage effectué en amont par Caml.

E. *Interface extérieure*

Bien que les questions d'interopérabilité entre langages n'aient pas été notre préoccupation initiale, nous avons mis en place un mécanisme d'intéropérabilité embryonnaire permettant d'appeler directement du code MSIL à partir de programmes Caml. En effet, Caml autorise des appels externes à des fonctions de bibliothèques écrites en C. Pour OCamIL, il est nécessaire de rediriger ces appels vers des bibliothèques .NET. Cela a été largement utilisé dans l'adaptation OCamIL de la bibliothèque standard Objective Caml, qui repose sur une bonne quantité de code C. Il a parfois été nécessaire de réécrire certaines fonctions (directement en code-octet, ou bien en C#), mais il est également possible d'appeler directement des fonctions de la bibliothèque de soutien .NET, comme dans l'exemple suivant, tiré du module `Sys` de la bibliothèque standard.

```
external il_getenv: string -> string =
   "string" "[mscorlib]System.Environment"
   "GetEnvironmentVariable" "string"

let getenv var =
  let s = il_getenv var in
    if s = "" then raise Not_found else s
```

La version OCamIL consiste en un soupçon de code Caml autour de la fonction `GetEnvironmentVariable` de .NET.

IV. Premiers résultats

A. *Distribution OCamIL 0.1*

La première version du compilateur OCamIL est disponible[10] dans sa version Windows. Nous avons adapté une grande partie de la bibliothèque standard de Objective Caml ainsi que la bibliothèque `graphics`.

Les traits fonctionnels, impératifs et objets[16] du langage ont été implantés, ainsi que le système de modules (foncteurs, compilation séparée).

B. *Performances*

Il y a une vraie difficulté à mesurer des temps d'exécutions sous les versions récentes du système Windows. On ne retrouve pas de commande permettant de mesurer les temps passés dans le processus "user" et dans les appels systèmes (à la manière de la commande `time` d'Unix). L'utilisation de `time` sous `cygwin`[17], une sur-couche Unix sous Windows, n'est pas satisfaisante dans la mesure où le temps calculé ne concerne que le *thread* principal. De plus le mécanisme de JIT introduit une différence notable entre la première exécution d'un programme et les suivantes. Pour ces raisons nous mesurons le temps «réel» passé dans l'exécution d'un programme .NET. Ce temps est une moyenne de trois exécutions au delà de la première.

Il est intéressant de comparer ces mesures par rapport au compilateur de code-octet d'Objective Caml qui nous sert de référence. Deux autres compilateurs vers .NET sont utilisés : F# qui est compatible avec le noyau impératif/fonctionnel d'Objective Caml et SML.NET qui compile le noyau de SML[6].

On peut grouper les tests suivants en deux groupes :

– les petits tests comme `Integrate` (calcul flottant), `Sieve` (listes d'entiers) et `DivEuclid` (calcul de termes).
– les tests correspondant à des applications comme `Boyer` (calcul de termes, appels de fonction), `KB` (calcul de termes, fonctionnel et exceptions) et `Nucleic` (calcul flottant et arbres). Ce dernier test sert de référence dans [7] pour tester une vingtaine de compilateurs de langages fonctionnels.

Les tests suivants ont été réalisés sur un Pentium IV 2,4GHz sous Windows XP. Ils ont été construits de manière à obtenir des temps proches d'une seconde pour le compilateur natif (`ocamlopt`). On cherche à comparer les résultats de OCamIL avec le compilateur de code-octet (`ocamlc`). En dehors des exemples très fonctionnels les temps de OCamIL sont proches de ceux d'`ocamlc` (facteur $1,5$). Par contre on obtient de moins bons résultats pour un programme très fonctionnel comme `KB` (facteur 25).

	ocamlopt	ocamlc	OCamIL	F#	SML.NET
Integrate	0,75	1,79	2,73	3,25	0.33
Sieve	0,73	4,07	6,59	2,87	1,19
DivEucld	0,93	13,1	18,6	4,94	1,69
Boyer	0,42	1,92	31,9	28,0	24,7
KB	1,07	7,30	170	216	209
Nucleic	1,14	6,57	7,53	3,79	1,04

Fig. 3. Tests de performance (temps réel en secondes).

Il est à noter que les compilateurs F# et SML.NET sont encore moins bons dans ce cas là que OCamIL. Cela provient des valeurs fonctionnelles et non pas du polymorphisme car ces deux compilateurs simplifient les contraintes de polymorphisme. F# compile vers une extension de MSIL, appelée ILX [8], introduisant la généricité. SML.NET à la manière de MLj [9] effectue au moment de l'édition de liens une analyse sur l'ensemble du programme pour le rendre monomorphe. Cette dernière optimisation, qui donne de bons résultats pour les programmes peu fonctionnels, empêche la compilation séparée.

La comparaison OCamIL et F# donne environ un facteur 2 à l'avantage de ce dernier. Le typage du code intermédiaire `Tlambda` de OCamIL n'est pas assez précis. Cela entraîne une allocation de structures de données plus importante que prévue (tableau d'objets) ce qui non seulement en ralentit l'accès mais nécessite des opérations de vérification de type dynamique coûteuses. Pour éviter cela une information supplémentaire provenant de la phase de typage du programme source devient néces-

[16] Voir la page WEB[10] de OCamIL pour le détail de l'implantation.
[17] www.cygwin.org

saire quand on compile vers une machine typée [10].

V. Travaux connexes

L'idée de pouvoir compiler des parties de programme issues de plusieurs langages vers la même plate-forme n'est pas nouvelle, y compris dans le monde fonctionnel typé. Plusieurs compilateurs vers C de ML ou de Scheme ont vu le jour au début des années 90 [11], [12], [13] facilitant l'interopérabilité à travers C. Néanmoins choisir comme cible une machine abstraite liée à un environnement d'exécution gérant la récupération mémoire et les ruptures de contrôle (exceptions) offre un bien meilleur niveau de sécurité à l'exécution, tout particulièrement dans le cadre d'asssemblage de code provenant de différentes sources. La JVM et le JRE (*Java Runtime Environment*) associé assurent ce niveau de sécurité. C'est d'ailleurs une des raisons du succès de la plate-forme Java. Avant la plate-forme .NET on pouvait déjà utiliser des langages fonctionnels typés statiquement (comme MLj [9]) ou dynamiquement (comme Bigloo [13]) compilés vers la JVM. Le code produit respecte les conventions de typage du code-octet de la JVM en facilitant l'interopérabilité de ces langages avec Java.

Cette idée de faciliter l'interopération entre langages n'est pas récente. Les langages de définition d'interface (*IDL*) tant pour CORBA[18] que pour COM[19] sont légions, y compris pour les langages fonctionnels, comme HDIRECT[14] pour Haskell[20] ou OCAMLIDL[21] pour Objective Caml. La plate-forme .NET est plus récente que celle de Java, son assembleur portable MSIL est plus riche du point de vue typage (Types-Valeurs) et contrôle (appels de méthodes via une indirection), tout en simplifiant le mécanisme de JIT. Dans les deux cas se posent des difficultés pour compiler des traits de programmation différents du modèle Java/C# comme les fermetures (ML, Scheme, Haskell), l'héritage multiple (Eiffel, Objective Caml) et les continuations Scheme.

VI. Conclusion

Nous avons montré avec cette première version de OCamIL la possibilité d'utiliser les langages à la Caml dans la plate-forme .NET avec des performances raisonnables tout en conservant la compatibilité du langage d'origine. L'étape suivante toujours dans la lignée de compatibilité est d'intégrer les outils spécifiques d'Objective Caml à cette plate-forme, comme les analyseurs syntaxiques (camlyacc, camlp4), les bibliothèques spécifiques ainsi que le *toplevel*. Ce dernier outil autorise un développement incrémental fort pratique pour le test immédiat des fragments d'un programme.

Du point de vue typage il est regrettable de perdre les informations inférées par le compilateur. Tout d'abord l'efficacité en souffre en encapsulant/désencapsulant inutilement les valeurs manipulées. Ensuite cette imprécision nuit à la qualité de mise au point comme l'inspection de valeurs en cours d'exécution mais aussi pour la précision d'un mécanisme de persistance. Le retypage du code intermédiaire d'Objective Caml s'apparente à la décompilation de code-octet [15] où seules certaines informations de typage sont nécessaires. L'idée est alors d'ajouter dans le code intermédiaire d'Objective Caml, principalement au niveau des fermetures, une information de type superficielle permettant une reconstitution précise.

L'utilisation de bibliothèques C#, ou de bibliothèques Objective Caml à partir de C#, doit être réalisée sans modifier la nature du langage fonctionnel/impératif/objet qu'est Objective Caml. Pour cela, nous étudions actuellement un IDL simplifié et extensible pour l'intersection des modèles objet d'Objective Caml et de C#.

Ces travaux en cours de réalisation ont comme principale motivation une meilleure diffusion de la programmation fonctionnelle statiquement typée. Notre volonté est d'offrir, y compris sur une plate-forme propriétaire, la possibilité de choisir le niveau d'abstraction offert par ces langages pour le développment d'applications.

Remerciement

Nous tenons à remercier Clément Capel pour son aide dans l'adaptation de la bibliothèque standard.

References

[1] X. Leroy, "The objective caml system release 3.06 : Documentation and user's manual," tech. rep., Inria, 2002. on-line version : `http://caml.inria.fr`.

[2] E. Chailloux, P. Manoury, and B. Pagano, *Développement d'Applications avec Objective Caml*. O'Reilly, 1st ed., 2000. on-line english version : `http://caml.inria.fr`.

[3] M.-V. Aponte, E. Chailloux, G. Cousineau, and P. Manoury, "Advanced programming features in objective caml," in *6th Brazilian Symposium on Programming Languages*, June 2002.

[4] A. Appel, "Runtime tags aren't necessary," *Lisp and Symbolic Computation*, 1989.

[5] P. Smacchia, *Pratique de .NET et C#*. O'Reilly, 1st ed., June 2003.

[6] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1991.

[7] P. H. Hartel and M. F. *et al*, "Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark," *Journal of Functional Programming*, vol. 6, pp. 621–655, Jul 1996.

[8] D. Syme, "ILX : Extending the .NET common IL for functional language interoperability," *Electronic Notes in Theoretical Computer Science*, vol. 59, no. 1, 2001.

[9] N. Benton, A. Kennedy, and G. Russel, "Compiling Standard ML to Java Bytecodes," in *Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming*, Sept. 1998.

[10] X. Leroy, "The effectiveness of type-based unboxing," in *Workshop on Types in Compilation*, June 1997.

[11] E. Chailloux, "An Efficient Way of Compiling ML to C," in *Workshop on ML and its Applications*, ACM SIGPLAN, June 1992.

[12] D. Tarditi, P. Lee, and A. Acharya, "No assembly required : Compiling standard ML to C," *ACM Letters on Programming Languages and Systems*, vol. 1, pp. 161–177, June 1992.

[13] M. Serrano and P. Weis, "Bigloo : a portable and optimizing compiler for strict functional languages," in *2nd*, (Glasgow, Scotland), pp. 366–381, Sept. 1995.

[14] S. Finne, D. Leijen, E. Meijer, and S. P. Jones, "H/direct : A binary foreign language interface for haskell," in *International Conference on Functional Programming*, 1998.

[15] Sébastien Ailleret, "Typage du bytecode Caml," in *Journées Francophones des Langages Applicatifs (JFLA02)*, Jan. 2002.

[18] `www.omg.org`
[19] `www.microsoft.com/com`
[20] `www.haskell.org`
[21] `caml.inria.fr/camlidl`