Université Paris 7 - Denis Diderot - UFR d'informatique
Année 2006 - 2007

Thèse
pour l'obtention du diplôme de
Docteur de l'Université Paris 7, spécialité informatique

Langages Fonctionnels, Typage et Interopérabilité : Objective Caml sur .NET

Raphaël Montelatici

Présentée et soutenue publiquement le
15 Mars 2007

Directeur de Thèse
Emmanuel Chailloux

Jury

M. Emmanuel Chailloux	*directeur de thèse*
M. Guy Cousineau	*président*
M. Jean-Marc Eber	*examinateur*
M. Michel Mauny	*rapporteur*
M. Bruno Pagano	*examinateur*
M. Christian Queinnec	*rapporteur*

Résumé

La plate-forme .NET est un environnement d'exécution moderne et répandu, reposant sur une machine virtuelle qui interprète du code-octet typé. Elle prétend être parfaitement adaptée à l'exécution de composants écrits dans une grande variété de langages de programmation et faciliter leur interopération.

En tant que langage fonctionnel statiquement typé avec polymorphisme paramétrique, Objective Caml présente des caractéristiques qui défient l'environnement d'exécution .NET et son système de types. Nous expérimentons ces difficultés dans un cadre pratique, par la conception et l'implantation de OCamIL, un compilateur complet pour Objective Caml qui produit du code-octet .NET vérifiable. Ses objectifs principaux sont la compatibilité et la possibilité d'interopérer.

Ce travail met à l'épreuve les capacités de la plate-forme .NET autant que l'adéquation de l'implantation officielle de Objective Caml dans un tel projet (celle-ci est conçue pour un environnement d'exécution dénué de types ce qui explique qu'elle élimine les informations de types assez tôt dans la chaîne de compilation). Nous examinons la représentation des valeurs Caml et comparons deux stratégies : la reconstruction et la propagation de l'information de typage manquante. D'autres choix de conception décrits ici illustrent le compromis entre efficacité d'une part et lisibilité/interopérabilité de l'autre.

Nous réalisons l'interopérabilité à l'aide d'un langage de description d'interface IDL qui construit un pont entre les deux systèmes de classes distincts utilisés par Objective Caml et l'environnement typé de .NET. Les bénéfices de l'interopération sont illustrés par des exemples non-triviaux.

Au chapitre des performances, OCamIL occupe une place respectable au sein des compilateurs de langages fonctionnels sur .NET. Nous comparons également les exécutables .NET avec les programmes Objective Caml originaux.

Abstract

The .NET platform is a modern, widespread execution environment, based on a virtual machine that interprets a typed bytecode. It claims to be perfectly suitable for running components written in many different programming languages and to allow a seamless interoperation between them.

Being a statically typed functional language with parametric polymorphism, Objective Caml has features that are quite challenging to the .NET runtime and its type system. We test the fit in a practical setting by designing and implementing OCamIL, a full-fledged compiler for Objective Caml producing verifiable .NET bytecode. It primarily aims at compatibility and interoperability.

This work questions the capabilities of the .NET platform as much as the adequacy of the official Objective Caml implementation (which is designed for an untyped runtime and discards type information early in the compilation chain). We discuss the representation of Caml values and compare two strategies: rebuilding and propagating the missing type information. Other design choices described here

illustrate the trade-off between efficiency and readability/interoperability.

We achieve interoperability by means of an Interface Description Language that closes the gap between the two distinct class systems of Objective Caml and .NET's underlying type system. We give non-trivial examples showing the benefits of interoperability.

As for performance, OCamIL proves competitive with other .NET compilers for functional languages. We also compare .NET executables with the original Objective Caml programs.

Remerciements

Tout d'abord, un grand merci à Emmanuel Chailloux, pour la qualité de son encadrement et sa grande disponibilité malgré ses nombreux engagements. Je lui suis reconnaissant de m'avoir sorti des griffes de la sémantique des jeux et des réseaux de preuves polarisés en me proposant un sujet bien différent mais tout aussi riche.

Merci à Michel Mauny et Christian Queinnec pour avoir accepté d'être les rapporteurs de cette thèse. Il est indéniable que leurs questions et leurs suggestions ont permis d'améliorer grandement la qualité du manuscrit final.

Je remercie Guy Cousineau, Jean-Marc Eber, Michel Mauny, Bruno Pagano et Christian Queinnec de m'avoir fait l'honneur d'accepter de faire partie du jury.

Cette thèse doit beaucoup à Bruno Pagano pour m'avoir encouragé à poursuivre ses recherches (sans lui OCamIL n'existerait sans doute pas) et Grégoire Henry pour sa collaboration (sans lui OJacaré.NET n'existerait sans doute pas).

Je salue les membres de l'équipe PPS qui tout en perpétuant un niveau d'excellence académique extraordinaire ont toujours pris le temps de partager leurs connaissances et de communiquer leur intérêt pour la recherche. Mes pensées vont particulièrement aux thésards de PPS : j'ai cotoyé plusieurs générations d'entre eux avec le même bonheur. Je n'oublie pas Odile Ainardi pour sa sympathie indéfectible et son talent à surmonter les défis administratifs. Michèle Wasse a elle aussi joué un rôle essentiel dans ces démarches.

Mon entière gratitude va à mes amis et ma famille, pour leur patience et leur soutien. Surtout, mes pensées les plus affectueuses vont à Julie.

Table des matières

Introduction

Interprétation [1] et compilation [2] fournissent deux modèles emblématiques de l'exécution de programmes informatiques, sans pour autant que les langages de programmation forment un univers manichéen partagé entre ces deux types d'implantation. Il faut se garder de confondre les langages et leurs implantations, puisqu'un même langage peut avoir plusieurs réalisations, mais il faut également prendre garde à ne pas opposer interprétation et compilation, dans une réalité bien plus complexe :

- les mêmes langages peuvent faire l'objet d'interprétation (qui a l'avantage de la portabilité et de la simplicité de la mise en œuvre) et de compilation (qui a l'avantage de la rapidité d'exécution) ;
- les modèles d'exécution des langages de programmation mélangent souvent les deux traits, et ce depuis les débuts de l'industrie logicielle.

Les machines virtuelles définissent une architecture d'exécution basée sur l'interprétation d'un pseudo-code (ou code-octet) ne correspondant, en général, à aucun équipement matériel concret. Le pseudo-code est produit par une phase de compilation à partir d'un langage de plus haut niveau destiné au programmeur. Ce fonctionnement permet un compromis entre efficacité (l'interprétation a lieu sur un jeu d'instructions simple et de plus bas niveau que le langage source) et portabilité (le langage compilé peut tourner sur toute plate-forme munie d'une implantation de la machine virtuelle). De plus la compilation JIT (*Just In Time*, également appelée « expansion de code ») peut se substituer à l'interprétation, donnant un modèle de livraison de code de la forme :

compilation de code indépendant de la plate-forme
↓
distribution
↓
compilation sur la plate-forme hôte

Cela illustre l'intérêt pragmatique pour une utilisation combinée d'interprétation et de compilation. D'autre part les plates-formes modernes munies de machines virtuelles permettent une administration poussée de l'exécution du code et de la

1. Un interpréteur prend en entrée un programme d'un langage donné et exécute les actions écrites dans ce langage sur une machine.

2. Un compilateur prend en entrée un programme d'un langage donné et le traduit dans un autre langage (le programme n'est pas exécuté). Le code produit peut à son tour être pris en charge par un interpréteur ou un autre compilateur. Dans le cas d'un compilateur de code natif, le code produit est directement exécutable par l'architecture cible.

mémoire : elles gèrent des processus légers (*threads*), permettent le chargement dynamique de code, offrent des services de sécurité, une interface avec le système, des outils de mise au point au niveau de la machine virtuelle et sont souvent munies d'un récupérateur automatique de mémoire.

L'utilisation de machines virtuelles est devenue très populaire avec les langages Java et C# (plus récemment) qui sont apparus avec leurs machines virtuelles respectives, la JVM et le CLR. Cependant il ne faut pas perdre de vue que les machines virtuelles existent depuis de nombreuses années pour toutes sortes de langages. Les plus connus dans les années 60 et 70 furent sans doute les langages SmallTalk, compilé dès ses premières versions vers une machine virtuelle (qui fit l'objet de spécifications avec la version SmallTalk-80), BCPL, compilé vers du « O-code » et UCSD Pascal, compilé vers du « P-code ». Ils furent rejoints dans les années 80 par Prolog (avec la machine virtuelle WAM) et d'autres langages fonctionnels. On peut citer la FAM (*Functional Abstract Machine*), définie par Luca Cardelli [16] et utilisée alors pour la compilation de ML, l'implantation CLISP de Common Lisp, LLM3 [23] conçue pour Le Lisp [24] (voir aussi [68] au sujet de Lisp) et la première implantation de Caml, la CAM (*Categorical Abstract Machine*) [28] puis la ZINC [52] utilisées pour Caml, Hugs pour Haskell...

L'ensemble des modèles de programmation est bien couvert par les machines virtuelles : impératif (P-code), fonctionnel (LLM3, FAM, CAM...), objet (SmallTalk, JVM, CLR) et logique (WAM). On pourra consulter [44] pour une comparaison de différents types de machines virtuelles. Comme c'est le cas pour Objective Caml, les implantations de langages utilisant une machine virtuelle peuvent fournir une alternative à base de code natif.

Les efforts de perfectionnement des machines virtuelles pour les rendre plus efficaces ont constitué un champ de recherche prolifique mais risquent de laisser de côté une perspective intéressante de ces machines : mettre en place un environnement d'exécution commun à plusieurs langages qui offre un contexte d'interopération simple et riche. Les machines virtuelles sont souvent dédiées à un seul langage ; c'est initialement le cas de la JVM, ensuite l'étendue de sa distribution a donné l'idée de l'utiliser comme cible de compilateurs d'autres langages que Java. La conception du CLR (la machine .NET) a tenu compte d'emblée de cette possibilité, proposant cette plate-forme pour les langages C#, Visual Basic.Net et C++ avec extensions gérées (puis J#, un dialecte de Java, peu de temps après).

L'interopération inter-langages ajoute un degré de liberté à la conception modulaire de logiciel : un projet peut être implanté au moyen de langages différents, pour répartir les tâches entre niveaux d'abstraction différents (un découpage courant est de confier l'architecture du projet à un langage de haut niveau alors que les segments optimisés ou chargés d'appels à des API systèmes sont écrits dans un langage de plus bas niveau) ou entre domaines où chaque langage excelle (par exemple une même application peut utiliser d'un côté C# pour son interface graphique et ses appels réseau et de l'autre Perl pour la manipulation d'expressions régulières). De plus l'interopération permet de fournir rapidement à un langage des bibliothèques qui lui font défaut, mais qui par ailleurs existent dans un autre langage.

L'intérêt de l'interopération inter-langages sur une plate-forme d'exécution gérée est accru par les garanties de sûreté qu'elle peut offrir (car l'interopération entre deux langages sûrs n'est pas elle-même automatiquement sûre).

Le travail de cette thèse vise à expérimenter l'adéquation entre :

– une machine virtuelle récente, répandue et offrant des perspectives intéressantes en termes d'interopérabilité,
– un langage fonctionnel possédant une large palette de constructions et un système de types riche.

Au niveau de la machine virtuelle, notre choix s'est porté sur la plate-forme .NET : la plus récente, ses capacités proclamées d'adaptabilité à des langages variés méritent d'être mises à l'épreuve. Du côté du langage nous avons choisi Objective Caml : il mêle des paradigmes fonctionnel, impératif et objet (ce dernier étant très différent des modèles de classes des langages Java/C#) et possède un langage de modules évolué. De plus dans une perspective de compilation la différence entre le système de types entièrement statiques de Caml et celui de la plate-forme cible, qui maintient les types à l'exécution, nous semble riche d'enseignements. La plate-forme .NET et le langage Objective Caml sont décrits dans le chapitre 1.

Notre expérimentation a consisté à mettre au point un compilateur .NET complet pour le langage Caml, baptisé OCamIL, et de profiter de ce développement pour évaluer à la fois la plate-forme .NET et l'implantation de référence du compilateur Caml dans ce contexte particulier, la discussion ayant pour objets principaux la confrontation des systèmes de types et les possibilités offertes par l'environnement .NET aux compilateurs de langages fonctionnels statiquement typés avec polymorphisme paramétrique.

Un volet important de notre travail a consisté à développer et évaluer un cadre permettant l'interopération de composants logiciels écrits en Objective Caml avec d'autres composants conçus dans n'importe quel langage disponible sur la plate-forme (même si C# fait figure de langage de référence, son système de types étant le plus proche de celui qui est intégré à la plate-forme).

Le premier chapitre de ce mémoire pose les principes du compilateur OCamIL après avoir introduit les principales caractéristiques de la plate-forme .NET et du langage Caml. Le chapitre 2 décrit en détails la compilation d'un programme Caml par le compilateur de référence (langages intermédiaires, représentation des valeurs) et examine le problème de la compilation vers le CLR sous l'angle du typage et de l'information de types, principale difficulté rencontrée dans le *front-end* du compilateur. Le chapitre 3 est consacré au *back-end* du compilateur OCamIL : il discute la représentation des valeurs Caml dans l'environnement .NET et l'émission de code. Le chapitre 4 fait un tour d'horizon des solutions disponibles pour Caml en matière d'interopérabilité et décrit la solution mise en œuvre dans le cadre du projet OCamIL, à savoir le compilateur de fichiers IDL « OJacaré.NET ». Le chapitre 5 illustre par quelques exemples des applications possibles de OCamIL et de OJacaré.NET.

Enfin le chapitre 6 est consacré à des tests de performances, compare différentes approches possibles au sein du compilateur OCamIL et le situe par rapport aux compilateurs standard de l'INRIA ainsi que d'autres compilateurs de langages de la famille ML sur .NET.

Chapitre 1

Préliminaires

Ce chapitre introduit le projet OCamIL mais décrit avant cela ce qui sert de toile de fond à ce travail : la plate-forme .NET et le langage Objective Caml.

Sommaire

1.1 Plate-forme .NET

D'abord connue sous le nom de Project 42, puis de COR, Lightning, COM+2.0, et NGWS (Next Generation Web Services), la plate-forme .NET a finalement retenu son appellation définitive quelques semaines avant son lancement au cours de PDC 2000 (Professional Developers Conference) à Orlando.

Mais que se cache exactement derrière cette dénomination très marketing? Nous reprenons dans les grandes lignes l'excellente présentation de la plate-forme .NET donnée dans [85].

1.1.1 Contexte et présentation

Au cours des dernières années l'acronyme .NET estampille toute la ligne des produits Microsoft. La gamme .NET se décline dans les domaines suivants :

- le développement logiciel (langages de programmation, outils de développement, plate-forme d'exécution),
- des technologies de service, pour le particulier ou les entreprises,
- la promotion de services Web commerciaux,
- la généralisation des équipements informatiques dans la vie quotidienne (cartes, téléphones mobiles etc. . .)

La présente thèse s'intéresse spécifiquement au premier des domaines cités ci-dessus, celui des langages de programmation, articulé dans le cadre de .NET autour de sa plate-forme d'exécution.

1.1.1.1 Objectifs

L'environnement .NET est manifestement issu des réflexions menées par Microsoft pour améliorer sa technologie COM (Component Object Model), voir la documentation COM en ligne [25].

Il s'agit toujours de fournir un moyen d'intégrer des composants logiciels, écrits dans des langages potentiellement différents. Tout au long de ce manuscrit, le terme *composant* désignera une unité déploiement d'une application, sous forme binaire[1].

La plate-forme .NET propose une nouvelle architecture cherchant à combler les différentes lacunes de COM, analysées selon les critères suivants :

1. Le mot *composant* est souvent utilisé pour des unités d'encapsulation au sein d'un langage, comme les classes pour les langages objet. Nous ne l'utiliserons pas dans cet usage.

Intégration. L'architecture COM a soulevé les problèmes suivants en matière d'intégration :

- Chaque composant doit se plier aux exigences du modèle COM, en particulier l'obligation d'écrire du code de « plomberie » nécessaire à l'interopération. Bien que répétitif, il doit être explicitement ajouté au code qui relève spécifiquement du composant.
- D'autre part les possibilités d'interaction sont limitées à des appels de méthodes. Il n'est pas possible de pousser plus loin l'intégration de langage en étendant une classe d'un composant à un autre par exemple.

La solution retenue par .NET est radicale : elle impose aux composants de partager un environnement d'exécution commun, et de manipuler un système de type commun. Cela entraîne naturellement la disparition du fastidieux code de plomberie et des conventions associées.

Déploiement. En matière de déploiement, le système de bibliothèques partagées des systèmes Windows (les DLL : *dynamic-link libraries*) a causé bien des soucis aux développeurs et aux utilisateurs. Les conflits entre DLL et les problèmes d'installation constitue un problème récurrent connu sous le nom *d'enfer des DLL* (*DLL Hell* en anglais).

La plate-forme .NET propose un système centralisé pour gérer l'installation et la résolution des dépendances aux bibliothèques. Pour une présentation détaillée de la solution proposée par .NET à ces problèmes on pourra consulter [65] en ligne sur MSDN.

Sûreté et sécurité.

- En ce qui concerne la sûreté du code, l'environnement .NET ne pouvait faire autrement que d'intégrer un certain nombre de mécanismes comme le typage, un système d'exceptions (ruptures de contrôle) et la récupération automatique de mémoire par un *garbage collector* (GC, glaneur de cellules ou ramasse-miettes). Ces avancées, popularisées par le langage Java, sont bien plus anciennes mais leurs mérites sont désormais universellement reconnus et elles sont présentes dans tous les langages modernes.
- Du point de vue de la sécurité, comme par exemple la mise en place de politiques d'autorisation d'exécution de code ou d'accès aux ressources, .NET propose une intégration au cœur même de son implantation et de ses langages.

La suite est consacrée à une description détaillée de l'environnement .NET. Les différents points ci-dessus seront à nouveau abordés.

1.1.1.2 Architecture

Afin de mettre l'accent sur une meilleure intégration, à la fois au système et entre composants multi-langages, le choix de Microsoft s'est porté sur une architecture proche de la machine Java.

La figure 1.1 montre une vue simplifiée des éléments constituant la plate-forme .NET.

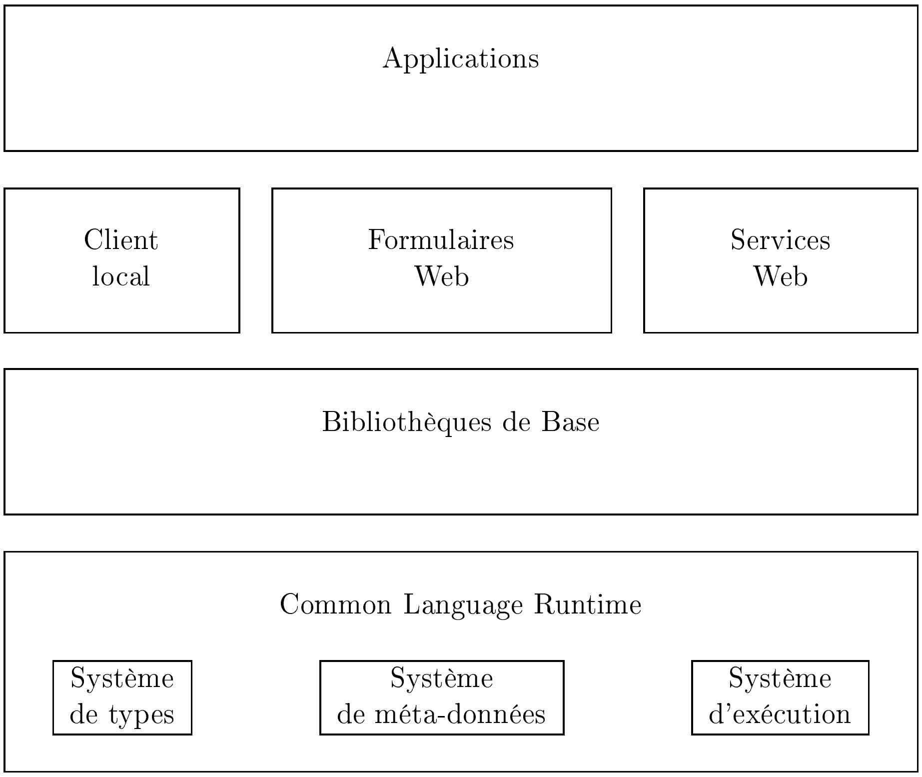


FIG. 1.1 – *Plate-forme .NET*

Le CLR (*Common Language Runtime*) est à la base de la plate-forme. Il est formé de trois composants principaux :

- Un système de types, conçu pour être capable de représenter les types et les opérations que peuvent réclamer la plupart des langages de programmation modernes, ceci afin d'offrir la possibilité à des langages variés d'être portés sur la plate-forme .NET. Nous serons amenés à critiquer cette affirmation au cours de cette thèse.
- Un système de méta-données. Celles-ci sont inclues avec les types dans le code compilé et sont utilisées par toutes sortes de programmes amenés à manipuler le code .NET, y compris l'environnement d'exécution lui-même. Nous décrivons les méta-données dans la section 1.1.3.
- Un système d'exécution qui repose sur une machine virtuelle à pile et fait tourner un code-octet portable.

La BCL (*Base Class Library*) est un ensemble de bibliothèques de classes mis à la disposition des exécutables .NET.

Cet ensemble sert de base à l'exécution de programmes compilés pour l'environnement .NET. Une application complète tire partie de la plate-forme à travers trois

types principaux de clients :
- des clients tournant localement sur un système d'exploitation prenant en charge l'architecture .NET,
- des formulaires Web, utilisés pour des applications Web articulées sur un serveur ASP.NET,
- des services Web, la technologie promue par Microsoft en matière d'appels distants de procédures.

1.1.2 Implantations

1.1.2.1 Plates-formes

La conception de la plate-forme .NET rappelle très fortement celle de Java. Cependant alors que Java est initialement prévu pour un seul langage, mais pouvant tourner sur des systèmes différents (comme le clame le slogan de Java : « Compile Once, Run Everywhere »), au contraire .NET a l'intention de recueillir de nombreux langages tout en étant principalement développée pour les seuls systèmes Windows.

Toutefois, il a été prévu de favoriser des implantations alternatives, afin d'assurer une meilleure diffusion de la plate-forme. Le CLR et les fondements de la bibliothèque de support ont fait l'objet d'une standardisation auprès de l'organisme ECMA [34]. Ce standard a servi de point de départ au développement d'autres implantations .NET.

On peut compter à l'heure actuelle quatre implantations principales, dont deux de Microsoft :
- la version officielle de Microsoft, qui tourne sous Windows. Pour l'instant téléchargeable librement, elle est sans doute amenée à devenir un composant à part entière du système d'exploitation Windows. Version la plus diffusée et profitant des tous derniers développements, ce sera notre implantation de référence.
- Rotor [81],une implantation open-source de Microsoft qui tourne sous Windows et Unix BSD. Rotor a été distribué pour des motifs de recherche et d'enseignement. Le JIT et le GC de Rotor sont des versions allégées de l'implantation commerciale, ce qui entraîne de moins bonnes performances.
- le projet Mono [33] de Ximian/Novell pour Linux, est Open Source. Ce projet est très abouti au niveau de l'environnement d'exécution mais ne propose pas encore une implantation complète de la BCL. En particulier il n'a pas été possible de se baser sur Mono lors du développement de OCamIL en raisons de lacunes initiales dans le domaine de l'API Reflection et de la gestion des assemblages signés.
- le projet Portable.Net de DotGNU est sous licence GPL. L'éventail de ses outils de développement est assez complet, mais son mécanisme de JIT n'est pas totalement opérationnel, ce qui nuit à ses performances.

Il existe aussi des variantes de la plate-forme .NET pour des architectures spéciales, comme le *compact framework* et les *smart devices extensions* qui ciblent des appareils mobiles. Ces environnements aux ressources contraintes nécessitent un environnement d'exécution allégé et des bibliothèques adaptées, ne fournissant pas tous les services des versions de bureau.

1.1.2.2 Langages

Dès le lancement de la plate-forme .NET, trois langages sont officiellement supportés par Microsoft : C# (sans doute le langage le plus proche des constructions implantées au sein de la machine virtuelle), Visual Basic.Net et C++ avec extensions gérées (au cœur duquel se trouvent directement implantés des mécanismes d'interopération entre code natif et code-octet). Ces trois langages ont rapidement été rejoints par J#, un dialecte de Java pour .NET.

Par ailleurs la firme a encouragé les initiatives d'adaptation d'autres langages à sa plate-forme, par le biais de ses laboratoires de recherche bien sûr, (SML [5] et F# [83] sont développés à Microsoft Research à Cambridge) mais aussi par l'incitation au développement par des entreprises ou des laboratoires d'universités, qui ont été suivis par des initiatives autonomes.

On peut trouver des compilateurs pour des langages aussi variés que : Ada A# [17], COBOL, Perl, Eiffel [78], Python, Pascal, Prolog P# [26], Fortran, SmallTalk, Oberon.

Du côté des langages fonctionnels, en plus de F# et SML.NET déjà cités, on trouve[2] : Lisp, Scheme (en particulier le projet Bigloo [73]), Haskell, Mondrian, Mercury [57], Ruby, Nemerle [60].

Ces différents projets ont bénéficié de la publication par Microsoft de spécifications de la machine virtuelle (un standard ECMA [34]) et de l'implantation *open source* Rotor [81].

Le développement de nouveaux compilateurs pour la machine .NET s'est également enrichi de l'adaptation de projets portant sur la plate-forme Java, comme Bigloo par exemple.

On pourra consulter une liste des différents langages portés sur .NET à l'adresse suivante : `http://www.dotnetpowered.com/languages.aspx`.

2. On pourrait inclure dans cette liste Python, Javascript et Perl qui comportent des traits de programmation fonctionnelle.

1.1.3 Système de types CTS et Méta-données

1.1.3.1 Le système de types CTS

L'environnement d'exécution manipule des données typées. Le système de types de .NET est judicieusement appelé CTS (*Common Type System*) : tout langage compilé devra s'y plier, ce qui a pour conséquences :

- De contraindre les implantations à faire rentrer les valeurs spécifiques de chaque langage dans le moule du CTS. On espère donc que ce système est suffisamment riche pour satisfaire des langages variés.
- De proposer un cadre commun, compris par tous, ce qui ouvre la porte à l'interopérabilité.

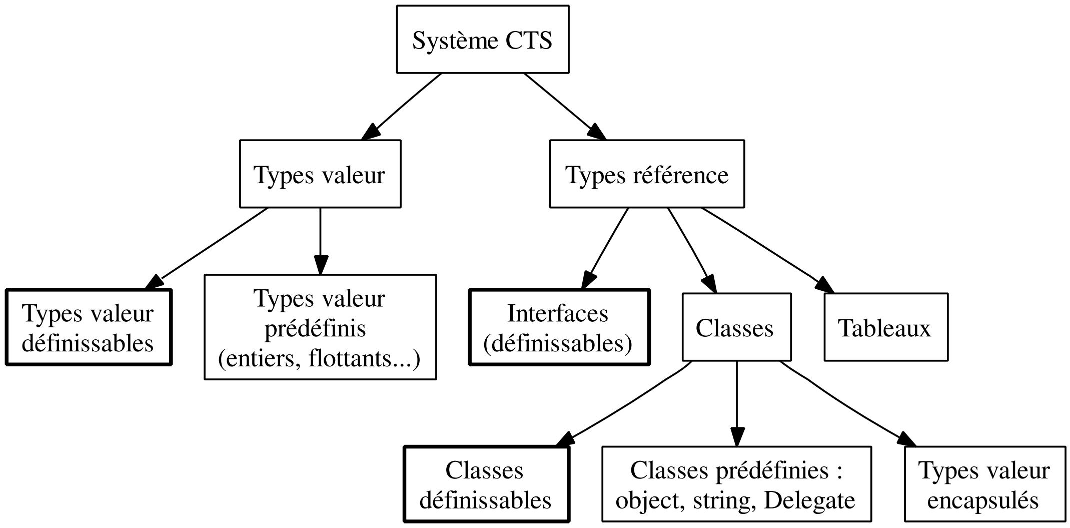


FIG. 1.2 – *Le système CTS dans les grandes lignes*

Modèle de classes et types références. Le modèle retenu pour le CTS est celui d'un système de classes similaire à celui de Java. Les types références regroupent les classes, les interfaces et les tableaux. Les valeurs des types références peuvent être `null` ou sont allouées dans le tas et soumises à la récupération automatique de mémoire. Elles sont passées par référence.

Le modèle présente les caractéristiques suivantes :

- Les classes sont organisées sur une hiérarchie d'héritage simple (pas d'héritage multiple) ayant pour racine la classe `System.Object` (avec pour alias `object`).
- Les classes comportent des méthodes et des champs. Ces membres peuvent être statiques (c'est-à-dire des membres de classe) ou d'instance.
- Les classes sont organisées en espaces de noms.
- On retrouve les attributs usuels concernant la visibilité et l'accessibilité des classes et de leurs membres.

Les types référence utilisateurs sont introduits par la déclaration de classes ou d'interfaces dans le code CIL engendré par la compilation du programme source. Les types sont organisés au moyen d'espaces de noms. En C# par exemple, la déclaration d'une classe de la forme :

```
namespace Graphics {
  class Point {
    public int x;
    public int y;
    public int norm() {...}
    ...
  }
}
```

introduit un type référence utilisateur `Graphics.Point`. Les espaces de noms peuvent être imbriqués.

Outre les classes, le CTS prend en charge les interfaces, les exceptions et les délégués :

- Les interfaces sont similaires à celles de Java et supportent l'héritage multiple (contrairement aux classes).
- Les exceptions sont également semblables à celles de Java. Ce sont des classes qui héritent de `System.Exception` et qui servent de base au mécanisme de rupture de contrôle pris en charge par l'environnement d'exécution.
- Les délégués, qui héritent de `System.MulticastDelegate`, sont une version *type-safe* des pointeurs de fonctions du langage C. Ils sont utilisés pour déclarer une classe et donc un type permettant l'encapsulation de méthodes ayant une signature donnée. La méthode encapsulée peut être statique ou d'instance, et la classe qui la définit n'a pas besoin de connaître l'existence du type délégué à l'avance (alors que pour appeler dynamiquement des méthodes d'une signature donnée par le biais d'un polymorphisme d'interface[3], il faut déclarer la relation d'implantation de l'interface dans toutes les classes pouvant être la cible d'un appel). Les délégués offrent d'autres services : ils peuvent maintenir une liste de méthodes à appeler et permettent des invocations asynchrones.

Les figures 1.3 et 1.4 permettent de comparer les implantations du design pattern « observateur » au moyen d'un délégué ou d'une interface. Dans les deux cas, l'implantation concrète de la méthode appelée n'a pas besoin d'être connue de l'appelant (ici la classe `Subject`). Cependant dans le cas des interfaces, l'implantation concrète doit déclarer a priori être un destinataire de l'appel (c'est-à-dire implanter l'interface `IObserver`) alors qu'avec les délégués il suffit d'exposer une méthode de la bonne signature (la liaison se fait de manière externe, au moment de la construction de l'objet délégué).

3. Le polymorphisme d'interface consiste à se servir d'une interface comme d'un type rendant compatibles entre elles toutes les classes qui implantent cette interface.

```
public delegate void DObserver(string msg);

class Subject {
  private DObserver notifyDelegate;

  public void registerObserver(DObserver n) {notifyDelegate = n;}

  public void notifyObserver() {
   if (notifyDelegate != null) notifyDelegate("hello");
  }
}

class ConcreteObserver {
  public void notify(string msg) { ... }
}

public static void Main() {
   ConcreteObserver o = new ConcreteObserver();
   Subject s = new Subject();
   DObserver deleg = new DObserver(o.notify);
   s.registerObserver(deleg);
   ...
}
```

FIG. 1.3 – *Design pattern « observateur » avec un délégué*

Types valeurs et opérations de (dés)encapsulation. Une nouveauté apportée par le CTS est la possibilité d'utiliser des types valeurs en plus des types références. Les types valeurs ne peuvent être `null` et ne sont pas alloués dans le tas, mais sur la pile. Il s'agit de types pour des valeurs structurées qui sont passées par copie. Ils permettent typiquement l'implantation d'unions ou de structures à la C, dans un cadre *type-safe*. Voici la liste des types valeurs prédéfinis (ils sont gérés de manière particulière par l'environnement d'exécution et des instructions CIL leur sont spécialement associées) :

- les booléens `bool`,
- les caractères `char`,
- les entiers signés : `int8`, `int16`, `int32`, `int64` et `native int`,
- les entiers non signés : `unsigned int8`, `unsigned int16`, `unsigned int32`, `unsigned int64` et `native unsigned int`,
- les flottants `float32` et `float64`.

Il existe aussi un type « référence typée » `typedref` utilisé à notre connaissance exclusivement pour l'implantation des types variants de Visual Basic.Net (nous n'en parlerons pas dans ce manuscrit, pour un exemple d'interopération entre Caml et Visual Basic, on pourra consulter [15]).

Les types valeurs utilisateurs (certains sont définis dans la BCL) se définissent à la manière de classes. Ils sont pourvus de champs et de méthodes, mais il n'y a pas d'héritage possible. Les types valeurs n'ont qu'un type possible a l'exécution. Ils servent à implanter les structures du langage C# (mot-clef `struct`) ainsi que les

```
interface IObserver {
   public void notify(string msg);
}

class Subject {
  private IObserver observer;

  public void registerObserver(IObserver o) {observer = o;}

  public void notifyObserver() {
   if (observer != null) observer.notify("hello");
  }
}

class ConcreteObserver : IObserver {
  public void notify(string msg) { ... }
}

public static void Main() {
   IObserver o = new ConcreteObserver();
   Subject s = new Subject();
   s.registerObserver(o);
   ...
}
```

FIG. 1.4 – *Design pattern « observateur » avec une interface*

énumérations (un cas spécialement géré par la plate-forme).

Tout type valeur est associé à un type référence, qui hérite de `System.ValueType` ou de `System.Enum` (qui lui-même hérite de `System.ValueType`). Par exemple le type `int` est associé à la classe `System.Int32` héritant de `System.ValueType`. Il est possible de passer d'une représentation à l'autre : les valeurs sont converties en objets alloués dans le tas via une opération de *boxing* (encapsulation). L'opération inverse s'appelle *unboxing* (désencapsulation). Ces deux opérations sont des instructions primitives de la machine virtuelle. Les classes correspondant aux types valeurs sont scellées (*sealed*) : elles ne peuvent pas être étendues par héritage (le mot-clé correspondant en Java est *final*).

Outre une différence sémantique dans la convention d'appel de méthodes, le choix entre types valeurs et types références se pose aussi en termes d'efficacité. Si d'une part la copie de valeurs peut être coûteuse, d'autre part les types valeurs ne sont plus administrés par le récupérateur de mémoire dont le travail est consommateur de ressources. Les données structurées de taille relativement faibles gagnent à être représentées par des types valeurs.

Héritage et sous-typage. La relation d'héritage du CTS définit une notion de sous-typage $\leq$ (les types concernés sont les types références et les types valeurs) :

– si τ est un type alors $\tau \leq \tau$,

- si la classe C' hérite de la classe C alors $C' \leq C$,
- si l'interface I' hérite de l'interface I' alors $I' \leq I$,
- si la classe C implante l'interface I alors $C \leq I$,
- si $\tau' \leq \tau$ alors $\tau'[] \leq \tau[]$.

Si $\tau' \leq \tau$ alors toute valeur de type τ' peut être utilisée en place et lieu d'une valeur de type τ. Les types valeurs ne sont en relation de sous-typage qu'avec eux-mêmes.

Lors de l'exécution toute valeur a un type *exact*, qui est le plus petit type acceptable pour cette valeur au sens de $\leq$. Les valeurs peuvent être considérées comme étant d'un sur-type du type exact, mais ce dernier est toujours connu de l'environnement d'exécution et déterminable dynamiquement. L'instruction `castclass` de CIL est utilisée pour transtyper une valeur le long de sa chaîne de sous-typage. L'échec est détectable car dans ce cas l'instruction `castclass` empile la valeur `null` (le modèle d'exécution de la plate-forme .NET est abordé à la section 1.1.4.1).

Le CTS présente d'autres caractéristiques que nous ne détaillerons pas ici, telles que les propriétés, les classes internes ou les pointeurs (pour l'interface avec le code natif). On pourra se reporter aux livres [79, 90, 85] pour de plus amples informations.

1.1.3.2 Interopération et CLS

Le CLR doit accueillir des langages différents et si le système de types CTS est pourvu de nombreux traits, il est clair qu'un langage donné ne va pas tous les appréhender. Dans un cadre d'interopérabilité il faut donc définir un sous-ensemble minimum sur lequel les langages peuvent raisonnablement se mettre d'accord. C'est ce que définit le CLS (*Common Language Specification*), un ensemble de règles qui limitent le système d'exécution et le système de types.

Sans vouloir entrer dans les détails, donnons deux exemples représentatifs :

- Il faut se plier à certaines conventions syntaxiques. Par exemple, le langage C# distingue les caractères majuscules et minuscules dans les identificateurs, mais pas Visual Basic.NET. Le CLS impose donc de ne pas compter sur cette distinction.
- Certains types ne sont pas connus de tous les langages. Le CTS distingue les entiers signés et non signés, mais certains langages ne connaissent pas les entiers non signés. Il faut donc limiter leur emploi.

Un composant sera conforme au CLS s'il suit ce type de règles. S'y contraindre n'est pas une obligation, les règles permettent simplement de définir un cadre commun pour l'inter-compréhension entre langages. L'ensemble des spécifications du CLS est détaillé dans le standard ECMA publié par Microsoft [34]. Il faut bien noter que ces restrictions ne s'appliquent qu'aux types intervenant à l'interface des composants logiciels, laissant possible d'utiliser la totalité du CTS en interne. Le CLS est supposé être assez complet pour que toute bibliothèque raisonnable puisse être écrite dans un style respectant ses consignes.

1.1.3.3 Assemblages et Méta-données

Assemblages. Les assemblages (*assemblies* en anglais) sont l'unité de base de déploiement .NET. Ce sont des exécutables ou des bibliothèques. Ils sont auto-descriptifs car en plus d'inclure le code proprement dit, ils recèlent un grand nombre d'informations gouvernant leur utilisation, comme les types exportés, les ressources utilisées, les dépendances etc…

- D'un point de vue client, un assemblage est un paquetage de types et de méthodes logiquement reliés les uns aux autres.
- Du point de vue du développeur, il s'agit d'une collection de fichiers, certains contenant du code .NET (on parle alors de modules) et d'autres (optionnels) incorporant des ressources diverses. L'un de ces modules et un seul doit contenir un *manifeste*, qui fonde l'assemblage en définissant ses composantes, ses dépendances et son identité (voir plus loin).

Le schéma 1.5 montre un exemple d'assemblage.

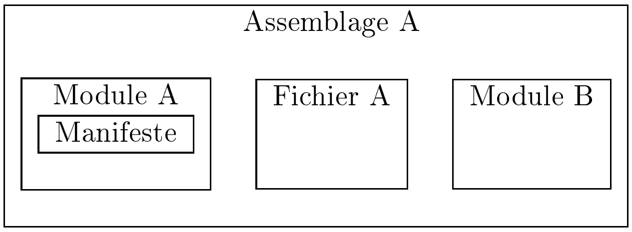


FIG. 1.5 – *Un exemple d'assemblage*

La grande force des assemblages par rapport aux bibliothèques traditionnelles réside dans les services qui y sont attachés :

- Ils sont munis d'une identité unique en déclarant de manière centralisée un nom, un numéro de version et des paramètres de localisation (une langue par exemple). L'authenticité d'un assemblage peut être garantie par un *nom fort*, c'est-à-dire une encryption par une paire de clefs publique et privée.
- Ils profitent des services de la plate-forme .NET en matière de déploiement. S'ils sont locaux à une application, leur présence dans un sous-répertoire du répertoire racine de l'application suffit à les intégrer à l'application. Il n'y a pas besoin de les enregistrer dans une base de registres comme c'est le cas pour les composants COM. S'ils sont partagés, ils doivent être placés dans un dossier virtuel central appelé GAC (*Global Assembly Cache*, ou cache global des assemblages), qui autorise ensuite une édition de liens dynamique depuis n'importe quel client.
- Ils décrivent les types exportés au moyen de méta-données, ce qui est la base de l'interopérabilité. Les méta-données sont décrites dans le paragraphe suivant.

Méta-données. Les méta-données sont des données sur des données. On peut les voir comme une forme enrichie des fichiers d'en-tête de C ou des interfaces compilées de Objective Caml.

Elles permettent la description des types[4] en référençant leur définition, en précisant leur disposition en mémoire par exemple, dans le but de permettre leur utilisation par un tiers.

Puisque les applications .NET peuvent reposer sur des assemblages écrits dans des langages différents, il faut prévoir la possibilité d'interfacer directement les langages au moyen de structures prises en charge au niveau de l'environnement d'exécution. Les méta-données sont émises directement dans le code produit avec les valeurs, afin de permettre toutes sortes d'instrumentalisations.

Nous pouvons indiquer à titre d'exemple quelques consommateurs de méta-données :

- **Le CLR.** Les méta-données sont utilisées pour les tâches de vérification et de gestion des politiques de sécurité (basées sur les types et les assemblages), la disposition en mémoire et l'exécution.
- **Le chargeur de classes.** Les références de types dans le code sont résolues à l'exécution, et les classes correspondantes trouvées et chargées au moyen des méta-données.
- **Le compilateur JIT.** Voir la section 1.1.4.2.
- **Différents outils**, comme les débogueurs, les dés-assembleurs, les profileurs, les environnements de développement intégrés exploitent tous les méta-données.

Incorporer les méta-données directement dans le code garantit par construction la cohérence de ces informations avec les types utilisés dans le code, et permet de résoudre les problèmes posés par les bibliothèques de types de COM.

La présence de méta-données accompagnant les valeurs au moment de l'exécution ouvre la porte aux mécanismes d'introspection et d'examen de types (*reflection* en anglais, nous utiliserons ce terme dans la suite par commodité). Il est possible, dynamiquement :

- d'inspecter des assemblages, des types, des membres de types,
- de charger des types et d'exécuter des méthodes,
- de générer dynamiquement du code, des types et des assemblages.

Les bibliothèques standard `System.Reflection` et `System.Reflection.Emit` permettent ces manipulations de méta-programmation, grâce à la persistance d'information avec les valeurs fournie par les méta-données.

Enfin, terminons ce tour d'horizon des méta-données en évoquant les attributs et les attributs utilisateurs (*custom attributes*). Il est possible d'incorporer des an-

4. Les méta-données sont omniprésentes : elles décrivent tout ce qui peut être décrit. Les informations contenues dans les assemblages (comme les dépendances, le numéro de version...) sont aussi des méta-données.

notations de son choix directement dans le code produit, à n'importe quel niveau (assemblage, classe, méthode, etc...) par le biais des méta-données.

Certains attributs sont standards et sont reconnus par l'environnement d'exécution, comme par exemple `DllImportAttribute`, qui permet de déclarer un prototype de méthode qui sera lié à une implantation externe (typiquement dans une bibliothèque non gérée compilée en code natif). D'autres sont à la discrétion du programmeur, et pourront être exploités spécifiquement par tel ou tel outil. Dans tous les cas, les attributs sont des instances de classes héritant de `System.Attribute`, classes qui peuvent être étendues par l'utilisateur.

1.1.4 Environnement d'exécution

Le CLR est capable de gérer l'exécution concurrente de plusieurs processus légers : ceux-ci disposent de plusieurs tas gérés et d'un espace mémoire à l'adressage partagé.

L'environnement est responsable de l'exécution des processus, à savoir : l'exécution du code lui-même, la gestion de la mémoire, du contrôle et des mécanismes d'appels, mais également du chargement des classes et des assemblages impliqués lors de l'exécution ainsi que différentes tâches relatives à la sûreté de l'exécution, comme la vérification de code ou l'application de politiques de sécurité.

Nous présentons dans la suite le modèle d'exécution proposé par le CLR.

1.1.4.1 La machine virtuelle et le code-octet CIL

Le CLR est articulé autour d'une machine virtuelle appelée VES (Virtual Execution System). Il s'agit d'une machine exécutant du code-octet basé sur un modèle de pile. Le bytecode .NET est appelé CIL (Common Intermediate Language). On parlera de code géré par opposition au code natif (que le CLR peut par ailleurs également prendre en charge).

État de la machine : états de méthodes et pile d'évaluation. Le mécanisme d'appels de méthodes est totalement pris en charge par le CLR, ce qui a pour conséquence de l'abstraire au niveau du code CIL. Ainsi les méthodes forment des unités de regroupement des instructions qu'il n'est possible de franchir au niveau du code géré qu'au moyen des quelques instructions d'appels disponibles dans le code-octet (il s'agit de `call`, `callvirt`, `calli`, `jmp`, `ret`, `throw` et `rethrow` : on trouvera la liste des instructions CIL en annexe).

L'état global de la machine virtuelle repose sur les différents états de méthodes, comme présenté sur la figure 1.6.

On peut modéliser un état de méthode par la conjonction de :
- Un pointeur d'instruction,
- Une pile d'évaluation,
- Un tableau d'arguments,
- Un tableau de variables locales (ce tableau est déterminé statiquement),

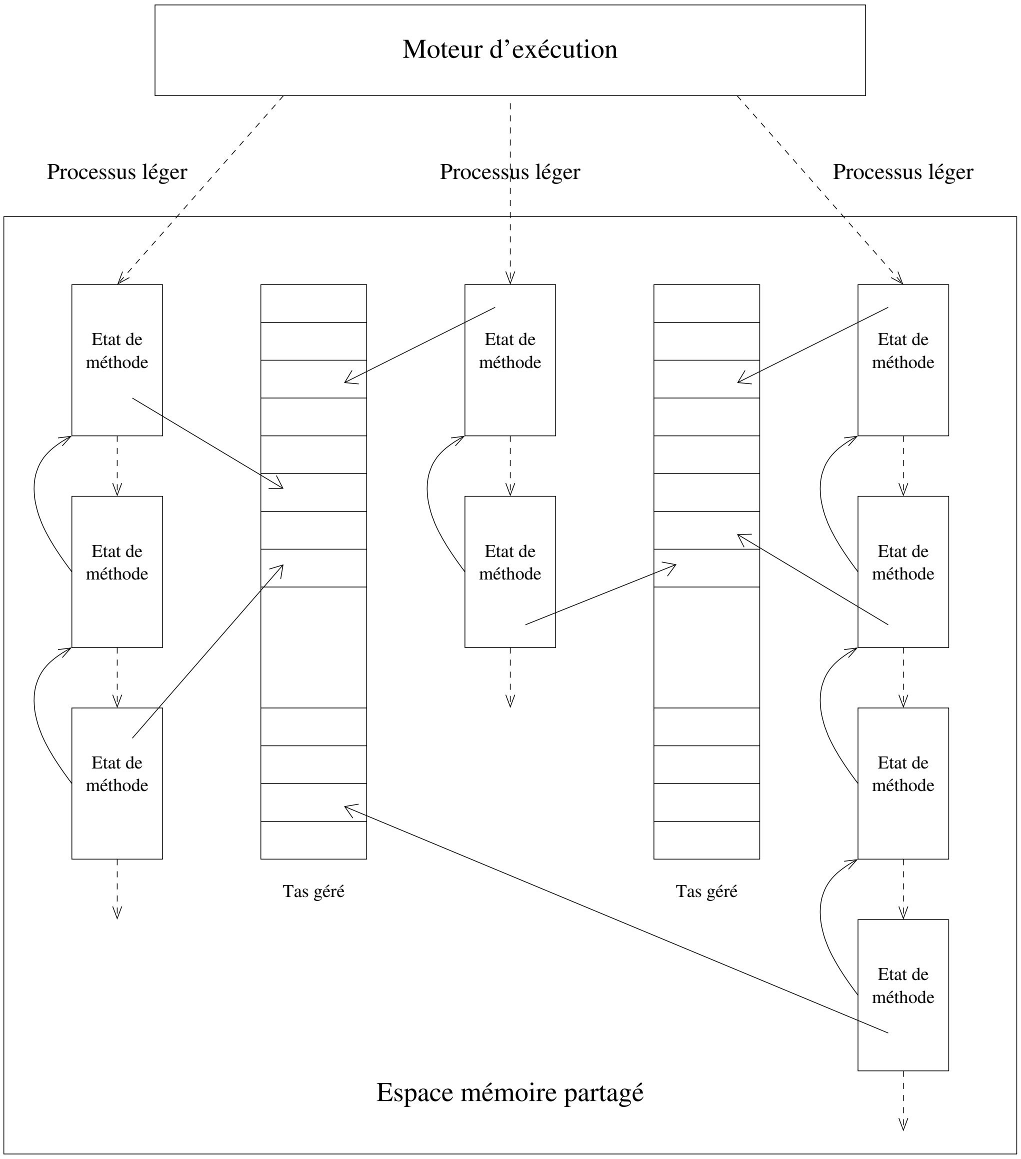


FIG. 1.6 – *État de la machine .NET*

- Une zone dynamique de mémoire locale,
- Un descripteur d'informations sur la méthode (contenant entre autres la signature de la méthode),
- Un descripteur de retour d'appel (permettant de restaurer l'état de la méthode appelante),
- Un descripteur de sécurité (utilisé par le CLR dans le cadre de ses politiques de sécurité).

Remarques :

- La pile, les tableaux d'arguments et de variables locales, ainsi la mémoire dynamique locale sont maintenus pour chaque méthode tant qu'elle ne termine pas. En particulier lorsqu'on passe d'une méthode à une autre avec une instruction `call`, l'état complet est restauré au retour de la méthode appelée (moyennant les ajustements de pile liés aux passage d'arguments et au retour de la méthode appelée bien sûr).

- Il est possible d'accéder aux tableaux d'arguments et de variables locales par leur adresse (exprimée sous la forme d'un pointeur géré).
- Les méthodes peuvent gérer un nombre variable d'arguments, au moyen de l'instruction `arglist` qui retourne une structure itérative permettant d'exploiter les arguments dynamiquement.
- La zone de mémoire locale est allouée dynamiquement au moyen de l'instruction `localloc`, le seul moyen de dés-allocation étant la terminaison de la méthode. Cette zone est adressable mais en général elle n'est pas utilisée de manière intensive comme peut l'être le tas (car sa taille ne peut pas être réduite au cours de la durée de vie de la méthode).
- La pile d'évaluation n'est pas adressable. Elle peut recevoir tous types de valeurs, y compris des types valeurs structurés (non boxés).

Tout emplacement qui sert au stockage ou au passage de valeurs, à savoir les arguments d'une méthode, les variables locales, les champs d'objets ou encore la pile, est muni d'un type. Cependant il faut distinguer les emplacements alloués statiquement comme les arguments, les variables locales ou les champs qui sont signés au moyen d'un type CTS explicite, des cellules de la pile pour lesquelles un nombre restreint de types sont définis. Ces types internes à la pile sont : les entiers 32 bits, 64 bits et natifs de la plate-forme hôte, un type de flottant F (de précision au moins supérieure à la précision des flottants natifs), un type O de « références gérées » et un type & de « pointeurs gérés »[5].

Les types de références et de pointeurs existent car des instructions CIL permettent de calculer l'adresse d'emplacements des valeurs afin d'agir sur elles par référence.

Dans ce contexte, on définit l'état typé de la pile à un point du programme comme la donnée conjointe de la profondeur de pile et l'ensemble des types CTS des valeurs sur la pile. Il est toujours possible de connaître le type exact d'une valeur sur la pile en suivant l'exécution pas à pas et on peut remarquer que l'un des soucis du compilateur JIT est de s'assurer que les bonnes conversions sont faites pour que les représentations internes à la pile n'affectent pas l'exécution d'un programme (que les bonnes opérations de conversions, de resserrage ou d'élargissement des valeurs soient insérées dans le code natif produit).

Dans le cadre de la vérification du code-octet, l'état typé de la pile est soumis à des contraintes vis-à-vis des différents chemins d'exécution possibles dans une méthode.

5. Les pointeurs non gérés, puisqu'ils doivent bien être pris en charge en même temps que le code natif, sont typés comme des entiers natifs. Pour plus de détails sur pointeurs, pointeurs gérés et références gérées on pourra se référer à [27]

Code-octet CIL. Les flots d'instructions CIL se déroulent méthode par méthode. Les instructions permettent la manipulation de la pile et des variables locales, comportent des opérations arithmétiques, de contrôle et de manipulation du modèle objet du CTS.

Chaque instruction attend un certain nombre d'arguments sur la pile, qui sont consommés, et dépose un éventuel résultat sur cette même pile. L'appel de méthode rentre dans ce cadre, par l'intermédiaire de l'instruction `call` et de ses variantes. Le détail des instructions CIL figure en annexe. Pour résumer on peut distinguer :

• *Les opérations numériques.* Le CIL comporte des opérations arithmétiques sur les entiers et les flottants. Les entiers peuvent être signés ou non, et leur taille varie de 1 à 8 octets. Les flottants sont en simple ou double précision. Il y a des instructions de comparaison, de conversion de types numériques, de test de dépassement arithmétique et des opérations booléennes bit à bit.

• *Les opérations de pile.* Dans cette catégorie se retrouvent tous les instructions permettant de déplacer une valeur vers ou depuis la pile. Cela peut se produire en relation avec les autres emplacements possibles pour les valeurs, figurant dans un argument ou une variable locale. Il est possible d'accéder à certains emplacement par leur adresse. On compte également les instructions de chargement des constantes numériques.

• *Les opérations de contrôle.* On trouve :
- Branchements conditionnels ou inconditionnels au sein d'une méthode.
- Appel de méthode : il peut se faire directement ou via une indirection. Les appels peuvent être virtuels (ayant recours au mécanisme de liaison tardive) ou non virtuels (liaison statique).
- Retour de méthode (instruction `ret`).
- Appels récursifs terminaux (*tail calls*) et sauts de méthode : dans certains cas il est utile d'appeler une méthode sans conserver le cadre d'activation de la méthode appelante. Cela est intéressant pour optimiser le code compilé pour une fonction récursive terminale par exemple.
- Mécanismes de rupture de contrôle (exceptions).

Le passage d'arguments d'une méthode à une autre peut se faire par valeur ou par référence (laissant aux instructions appropriées le soin de calculer l'adresse de l'emplacement d'une valeur ou de charger une valeur par indirection).

Une alternative existe, spécialement implantée pour Visual Basic.NET : le passage par référence typée. Une référence typée est une valeur contenant à la fois l'adresse d'une valeur et son type. Cela permet un accès par référence pour lequel le type de la valeur est consultable dynamiquement.

• *Les opérations de mémoire et opérations sur les objets.* Cela regroupe la création d'instances de classe, l'accès aux champs d'une classe, les tests dynamiques de types, le *downcast* (consistant à modifier le type dynamique d'un objet vers un autre type possible pour cet objet, plus précis dans la hiérarchie de classes) et les opérations de boxing/unboxing. On peut également compter dans cette catégorie les opérations sur les tableaux.

Remarque valant pour toutes les instructions : du point de vue des types, on trouve,

- des instructions polymorphes, comme `dup` ou `add`,
- des instructions paramétrées par des types de base, comme `ldc.i4` (le type fait partie de l'instruction, pour des types différents, on aura des *opcodes*[6] différents),
- des instructions paramétrées par des références à des méta-données décrivant le type CTS manipulé[7], comme `isinst MyNamepace.MyClass` (ici le type suit l'instruction dans le flot des opcodes).

1.1.4.2 Chargement et exécution

La figure 1.7 illustre les différentes étapes menant de la compilation du code source à l'exécution par l'environnement .NET. Les boites grises sont des éléments de la plate-forme.

Les assemblages générés par l'édition de lien prennent la forme de fichiers au format PE (*Portable Executable files*) : ceux-ci rassemblent code-octet, code natif, méta-données et éventuellement des données de ressources.

Chargement de classes. Les assemblages pris en entrée du chargeur de classes ont pu être produits par des compilateurs de langages différents. À coté du code octet, le code natif est également pris en charge, et les deux peuvent être produits par un même compilateur (c'est le cas pour le C++ avec extensions gérées par exemple).

Lors de l'exécution d'un assemblage, le CLR est chargé de résoudre les références aux classes internes ou externes à l'assemblage. Cette résolution passe par l'identification des classes et de leur source par l'intermédiaire de méta-données : références de classes et références aux assemblages tiers. Les méta-données intègrent des informations de version et incorporent un mécanisme d'authentification des sources grâce à la signature par les noms forts. L'emplacement physique des assemblages à charger est déterminé à l'aide des méta-données et de réglages administratifs de la plate-forme hôte.

Selon le niveau de confiance accordé au code chargé, le mécanisme de vérification peut être appliqué. De son côté le code natif doit toujours être digne de confiance car il ne peut être vérifié. L'exécution a lieu moyennant l'application des politiques de sécurité, dont certaines clauses s'expriment de manière dynamique.

6. Un opcode désigne l'octet ou les octets représentant physiquement l'instruction en mémoire.

7. En pratique, les références aux méta-données sont implantées au moyen de clefs de quelques octets insérées dans le code, qui pointent vers la table des méta-données utilisées par chaque assemblage. On appelle ces clefs des *jetons de méta-données*.

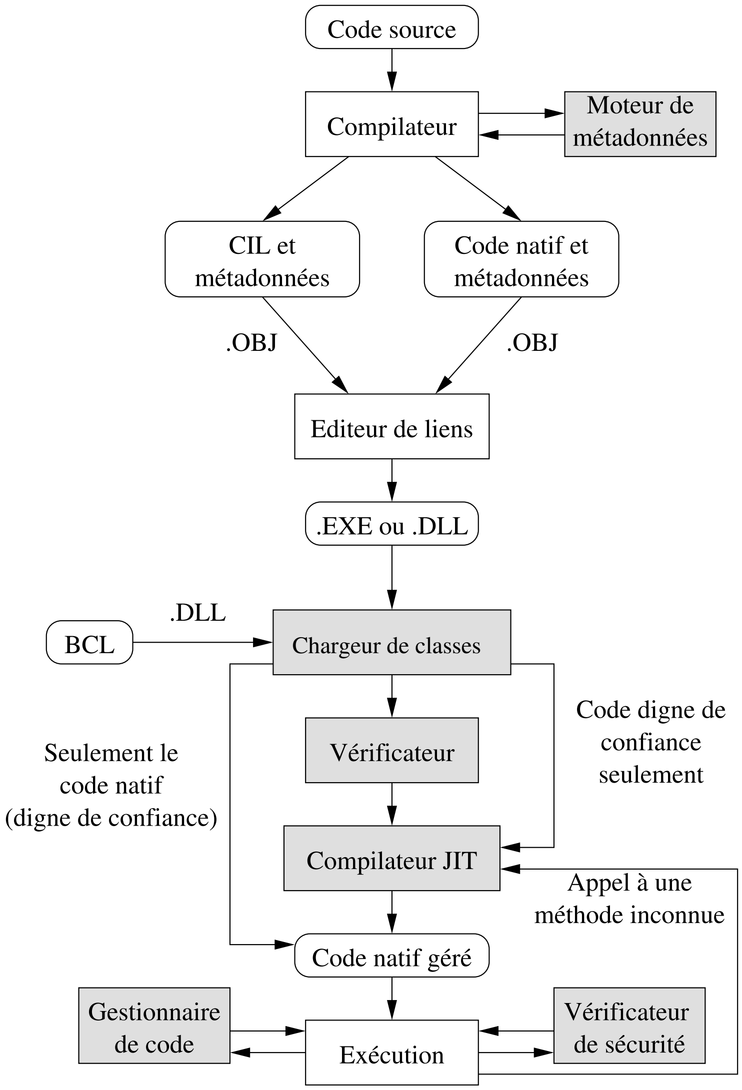


FIG. 1.7 – *Compilation et exécution*

Compilation *Just In Time*. Dans l'univers des machines virtuelles, il est possible d'interpréter le code-octet ou bien de le compiler afin d'exécuter du code natif. Cela se nomme la compilation Just In Time (JIT) puisqu'elle a lieu au dernier moment (à l'exécution) sur l'architecture concrète hébergeant la machine virtuelle.
En résumé :

- On préfère distribuer le code-octet, portable, que du code natif précompilé.
- La compilation JIT permet ensuite d'attendre de connaître la plate-forme hôte afin de compiler efficacement en tenant compte de ses caractéristiques.

Les JVMs utilisent différentes heuristiques choisissant au cas par cas si une méthode doit être interprétée ou bien compilée juste à temps. Car si une méthode ainsi compilée s'exécutera plus rapidement pour tous les appels ultérieurs, le temps passé (lors de l'exécution du programme) à effectuer la compilation juste-à-temps est loin d'être négligeable.

Du côté de l'architecture .NET, le choix est simple : chaque méthode CIL est compilée en code natif lors de son premier appel. Il s'agit donc d'un JIT systématique.

1.1.4.3 Questions de sûreté

Code géré et typage. Les programmes *type-safe* ne référencent que la mémoire qui a été allouée pour eux et n'accèdent aux objets qu'à travers leur interface publique. Cela autorise les objets à partager un même espace d'adressage de manière sûre et fait en sorte que les garanties prévues par les interfaces des objets ne sont pas contournées.

Il faut toutefois un moyen de vérifier cette propriété sur les programmes chargés par le CLR, alors même que ces programmes proviennent de sources éventuellement non sûres, et ont été compilés à partir de langages différents, proposant des mécanismes de vérification statique plus ou moins élaborés.

Le modèle proposé par les langages statiquement typés tel que Objective Caml ne se transpose pas dans le cadre des machines virtuelles multi-langages. Cependant, la présence de méta-données (et donc de types) directement incluses dans le code géré permet de répondre à ce problème. En revanche les sections de code non géré doivent faire l'objet d'une confiance totale.

Le processus de vérification du bytecode consiste à vérifier la cohérence des méta-données (par exemple les noms de types eux-mêmes, qui doivent correspondre, mais aussi la manière de les instancier en mémoire, en précisant l'alignement, la convention d'*endianness*...), appliquer un certain nombre de critères limitant l'utilisation d'instructions critiques et faire une analyse des flots d'exécution au sein du code pour garantir des propriétés de cohérence de pile (ainsi l'état typé de la pile tel que défini plus haut doit être identique en tout point du programme vis à vis des différents chemins d'exécution qui peuvent y mener). Ce processus ne s'applique qu'à du code géré : le code natif n'est jamais vérifiable.

Au bout du compte, on peut classer le code CIL géré de la manière suivante :

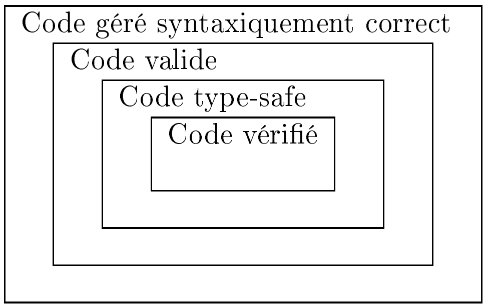


FIG. 1.8 – *Les différents niveaux de correction du code CIL géré*

- Code géré syntaxiquement correct.
- Code géré valide. C'est du code pour lequel le compilateur JIT est capable de produire une représentation native. Du code invalide correspond à des instructions illégales dans leur contexte (c'est le cas par exemple si une instruction de branchement `br` pointe à l'extérieur du code de la méthode dans laquelle elle apparaît).
- Code géré *type-safe*. Cette propriété peut être vraie pour du code utilisant de manière sûre des opérations de nature non sûre.
- Code géré vérifié. C'est un sous-ensemble du code type-safe pour lequel cette propriété est prouvable algorithmiquement.

La classe de code vérifié dépend de l'algorithme de vérification choisi. L'important est que celui-ci soit conservatif, ne laissant passer que du code type-safe. Cependant la sûreté par le typage n'est pas une propriété décidable par un algorithme et il faut se satisfaire d'un algorithme conservatif qui rejettera parfois du code type-safe : il faut arbitrer entre précision et coût de l'algorithme. L'algorithme de vérification implanté dans le CLR réalise ce compromis entre exhaustivité et vitesse d'exécution.

Au sujet du code non-géré. Il est également possible de produire dans les assemblages .NET du code non-géré, c'est-à-dire du code natif (qu'il ne faut pas confondre avec le code géré non-vérifiable). Le code non-géré tourne en dehors de tout contrôle de l'environnement d'exécution : en particulier il n'y a pas de ramasse-miettes et les structures de données ne sont pas munies de type CTS.

Certaines instructions CIL permettent de faire le lien entre le monde géré et le monde natif : manipulation de pointeurs, appel de fonction par indirection, manipulation de blocs de mémoire. Bien entendu ces instructions ne sont pas vérifiables.

Le seul langage à proposer la génération de code non-géré et même un mélange entre code géré et code non-géré, est Visual C++ avec extensions gérées (des langages comme F# proposent le choix entre générations de code géré vérifiable et géré non-vérifiable, ce qui est totalement différent).

Autres mécanismes de sécurité. Au delà de la sûreté par le typage, le CLR applique des politiques de sécurité permettant de ne donner l'accès à certaines méthodes que sous certaines conditions, statiques ou dynamiques, sur la méthode appelante. Ces conditions exprimées sous la forme d'objets et de méta-données de sécurité, sont éventuellement contrôlables de manière administrative sur la plate-forme hôte.

Au chapitre de la sécurité on peut également citer le mécanisme de ramasse-miettes mis en œuvre par le CLR, qui évacue tous les problèmes liés à la gestion de la mémoire par le programmeur.

1.1.5 Utilisation de la plate-forme pour la compilation d'un langage fonctionnel

Nous concluons cette présentation de la plate-forme .NET par un tour d'horizon des moyens mis à la disposition d'un compilateur de langages fonctionnels.

1.1.5.1 Apports de la plate-forme

Au premier coup d'oeil la plate-forme .NET semble surtout adaptée aux paradigmes impératif et objet. Toutefois, certaines caractéristiques sont très utiles pour l'implantation de langages fonctionnels comme Caml :

Appels récursifs terminaux (*tail calls*). Les langages fonctionnels conduisent à un style de programmation reposant de manière intensive sur la récursion. Cela peut provoquer des profondeurs de pile d'appel très importantes à l'exécution, qui peuvent être évitées grâce à l'identification d'appels récursifs terminaux, pour lesquels le cadre d'activation d'une fonction peut être abandonné d'un appel récursif à un autre, évitant ainsi des empilements inutiles.

Le CLR gère lui-même le mécanisme d'appels de méthodes et il est heureusement prévu de réaliser des appels récursifs terminaux (au moyen du préfixe `tail.` ou des instructions `jmp`). Remarquons que cette possibilité est absente de la machine Java.

Récupération automatique de mémoire. Les langages fonctionnels, à l'instar de la plupart des langages modernes (avec Java et C# en première ligne), reposent sur un mécanisme de récupération automatique de la mémoire.

L'environnement d'exécution de Objective Caml incorpore son propre GC, et il est pratique pour tout compilateur ciblant la plate-forme .NET de pouvoir s'appuyer sur le GC de celle-ci.

1.1.5.2 Lacunes de la plate-forme

Cependant un certain nombre de détails laissent penser que les langages fonctionnels n'ont pas été au coeur des considérations lors de l'élaboration de la plate-forme .NET.

Fermetures. Il n'y a pas de support natif pour les fermetures, une structure de donnée incontournable dans l'implantation des langages fonctionnels. Une extension développée dans les laboratoires de recherche de Microsoft, appelée ILX [82], tente de combler cette lacune mais ne fait partie de l'environnement .NET officiel. Ce sujet est de nouveau évoqué à la section 1.3.1.2.

Types algébriques et polymorphisme. Les types algébriques chers au langages fonctionnels n'ont pas de représentation directe dans le système de types CTS, et le polymorphisme d'interface des langages objets ne correspond pas au polymorphisme paramétrique de ces langages. Il faudra donc dans les deux cas trouver un moyen raisonnablement efficace d'implanter ces traits essentiels.

En ce qui concerne le polymorphisme paramétrique, les Generics [50, 92], à l'origine une extension expérimentale et par la suite ajoutée au CLR dans sa version 2.0, propose une solution partielle (voir la section 1.3.1.2 pour plus de détails).

1.1.5.3 Caractéristiques utiles au développement

Indépendemment du type de langage développé sur la plate-forme, celle-ci offre un certain nombre de services qui permettent la mise au point rapide d'un compilateur.

Génération de code. Il n'est nullement nécessaire de générer des fichiers au format PE, du moins lors de la phase de mise au point. Si certains choisissent de générer du code C# qui sera ensuite compilé en code-octet par le compilateur C# `csc`, la plupart des projets génèrent du code-octet CIL sous forme de mnémoniques, compilée par la suite au moyen de l'assembleur de code CIL `ilasm`. Ces deux compilateurs sont fournis avec la plate-forme.

D'autre part la BCL comporte de nombreuses méthodes d'émission de code, en particulier toute l'API `Reflection.Emit`, ainsi que des méthodes permettant d'accéder aux méta-données. Cette approche est envisageable dans le cadre d'un compilateur bootstrappé, dont le code aura naturellement accès à ces services.

Débogage. La mise au point d'un compilateur est facilité par les systèmes de garde-fous, de rattrapage et d'analyse d'erreurs que fournit la plate-forme. Ainsi du fait même de la présence des types à l'exécution et de la gestion native des exceptions, il est facile d'avoir des indications lors d'un crash du programme (que ce soit les programmes engendrés par le compilateur en phase de mise au point ou bien même le compilateur bootstrappé un peu plus tard lors du cycle de développement, voir la section 5.2.1.1).

Les outils suivants, fournis avec la plate-forme, sont également très utiles :
- le dés-assembleur de code CIL `ildasm`,
- le vérificateur de code CIL `peverify`,
- le débogueur `cordbg`.

1.2 Objective Caml : langage et compilateur

Nous donnons ici une présentation succinte du langage Objective Caml en insistant sur le système de type et l'architecture du compilateur.

1.2.1 Les types et les valeurs Objective Caml

Le langage Objective Caml est un langage typé statiquement, avec inférence de types. Cela signifie que le programmeur écrit habituellement son code sans annotation de types : lors de la compilation, l'algorithme d'inférence de type de Caml calcule les types que doivent avoir les termes et sous-termes du programme. Si cette étape ne décèle pas d'erreur de types, la compilation se poursuit et produit en principe un programme sûr. Il est important de noter que les types sont exploités de manière purement statique et que le code produit est dépourvu d'indications de types.

Le système de types de Objective Caml est assez riche, nous le décrivons en nous inspirant du manuel du langage [54], auquel le lecteur pourra se référer pour plus de détails. Il convient de distinguer définitions de types et expressions de types.

1.2.1.1 Définitions de types

Les expressions de types Caml font référence à deux sortes de types : des types anonymes dont la définition est implicite et des types nommés pour lesquels le programmeur (ou le noyau Caml pour les types de base) donne une définition préalable à leur utilisation. Cette section est consacrée au langage de définition de types.

Grammaire des définitions de types. Dans la grammaire de Objective Caml, une définition de types s'exprime comme :

```
type-definition ::= type typedef  { and typedef }
typedef ::= [type-params]  typeconstr-name  [type-information]
```

Comme l'indique la première ligne, il est possible de définir des types mutuellement récursifs. De plus chaque type peut être paramétré par une variable de types ou un n-uplet de variables de types (c'est ce qui est derrière `type-params`, non détaillé ici).

Pour le reste, `typeconstr-name` est le nom du type défini par le programmeur et `type-information` est constitué de deux éléments optionnels : une équation de type et une représentation de types. Il y a donc quatre possibilités selon la présence d'équation de type et de représentation de type :

– **Définir un type abstrait**, par exemple : `type t`. Sans équation ni représentation, on définit un type abstrait. Il sera incompatible avec tout autre type. Dans une interface, cela permet de masquer l'implantation réelle du type. Parfois l'implantation est elle-même abstraite : cela permet de définir un type opaque pour l'environnement Caml mais exploité par des primitives externes codées en C : dans la bibliothèque standard, c'est par exemple le cas des types `in_channel` et `out_channel`.

Les types de bases tels que `int`, `float`, `'a array` sont des types abstraits (mais gérés de manière particulière par le compilateur).

- **Définir une abréviation de type**, par exemple : `type t = int`. Cela définit un alias de type. Cela est souvent utilisé pour préciser une implantation concrète dans un fichier d'implantation alors que celle-ci reste abstraite dans le fichier d'interface.
- **Définir un nouveau type variant ou enregistrement**. C'est la partie la plus créative des définitions de types en Caml, permettant au programmeur de définir des types algébriques riches. Citons des exemples simples, pour un type variant : `type 'a t = A of 'a * 'a t | B`, ou `type t = {x:int; y:int}` pour un type enregistrement. L'expressivité du langage repose en grande partie sur les types algébriques : les valeurs de ces types peuvent être directement construites ou inspectées, et se soumettent au filtrage de motifs.
- **Définir un type variant ou enregistrement réexporté :** par exemple en reprenant la définition précédente : `type new_t = t = {x:int; y:int}`. Cela définit un nouvel alias de type, tout en réaffirmant la représentation de ce type (elle doit correspondre parfaitement à la représentation du type initial).

Les types définis par le noyau Caml. La figure 1.9 liste les types prédéfinis par le langage Caml.

Nom	paramétré	abstrait	définition
`int`		$\surd$	`type int`
`char`		$\surd$	`type char`
`string`		$\surd$	`type string`
`float`		$\surd$	`type float`
`bool`			`type bool = false | true`
`unit`			`type unit = ()`
`exn`			défini comme variant sans constructeur
`array`	$\surd$	$\surd$	`type 'a array`
`list`	$\surd$		`type 'a list = [] | (::) of 'a * 'a list`
`option`	$\surd$		`type 'a option = None | Some of 'a`
`lazy_t`	$\surd$	$\surd$	`type 'a lazy_t`
`nativeint`		$\surd$	`type nativeint`
`int32`		$\surd$	`type int32`
`int64`		$\surd$	`type int64`

FIG. 1.9 – *Types prédéfinis par le langage Caml*

Les définitions de types variant données dans ce tableau seraient parfaitement acceptables dans un module utilisateur, mises à part les conventions syntaxiques imposant aux constructeurs de variants d'être des chaînes de caractères alphanumériques commençant par une majuscule). Cependant les définitions inclues dans le noyau du langage sont centrales et sont prises en charge de manière spécifique par l'implantation pour la majorité d'entre elles.

Au sujet des noms de types. Les modules et sous-modules définissent des espaces de noms qui permettent de qualifier les types. Les noms de types associés aux définitions doivent être uniques au sein d'un même chemin de type (c'est-à-dire qu'on ne peut pas avoir dans le même programme Caml deux définitions de types aboutissant au même nom pleinement qualifié).

1.2.1.2 Expressions de types

Les expressions de types annotent les termes et les sous-termes formant un programme Caml au cours de l'inférence de types [8]. Les expressions de types sont construites à partir d'éléments natifs du langage (types de base de Objective Caml et constructions telles que types produits , types flèches, listes, tableaux...) ou d'éléments définis par le programmeur (types variant, enregistrements ...) obtenus en utilisant les moyens décrits à la section précédente.

Nous présentons informellement la grammaire des expressions de types :

Variables de types : à la base du polymorphisme, les variables de types, qu'elles soient nommées (`'a`, `'b`, ...) ou anonymes (`_`), peuvent s'instancier en n'importe quelle autre expression de type. Ce sont des terminaux de la grammaire.

Elles figurent également en paramètres des constructeurs de types polymorphes (voir plus bas).

Types construits : autres terminaux de la grammaire, les constructeurs de types forment la substance indivisible des expressions de types. Ce sont des références (par leur nom) aux types définis, qui peuvent être abstraits (comme `int` ou `'a array`) ou algébriques (comme `'a list`). Ils peuvent être polymorphes, comme le montrent précisément les tableaux et les listes. Dans ce cas, les paramètres de types peuvent faire l'objet d'instanciations.

Types fonctionnels : c'est le type flèche `->` qui correspond à l'abstraction du λ-calcul. Le langage Objective Caml propose d'étendre le paradigme fonctionnel pur au moyen d'arguments étiquetés ou optionnels, qui se ramènent de toutes façons à des types flèches traditionnels.

Exemples de types flèches : `int -> int` ou `(int -> int) -> int`. Ce dernier est bien sûr différent de `int -> (int -> int)`, qui par convention pourra être noté `int -> int -> int`.

Types n-uplets : le constructeur `*` permet de former le produit de n expressions de type (avec $n \geq 1$).

Exemples : `int * (int -> int)` ou `'a * 'b * int`. Noter que ce dernier est différent de `'a * ('b * int)` et de `('a * 'b) * int`.

8. Le programmeur peut également écrire des expressions de types dans les signatures des modules ou dans des coercitions explicites de types, afin d'imposer des contraintes à l'algorithme d'inférence de types.

Types variants polymorphes : ces expressions correspondent à des types variant sans nom et extensibles. Le principe est d'énumérer dans l'expression de types elle-même l'ensemble des constructeurs possibles dans ce type, de manière exacte ou allusive (indiquant alors quels constructeurs sont *au moins requis* et/ou quels constructeurs sont *au plus autorisés*). Les variants polymorphes, au contraire des variants classiques introduits précédemment, ne se basent pas sur une définition de types.

Par exemple l'expression `` [> `A | `B of int] `` fait référence à tout type variant ayant au moins les constructeurs `` `A `` et `` `B `` indiqués, l'expression `` [< `A | `B of int] `` à tout type variant ayant au plus ces deux constructeurs. La conjonction des deux définit un type exact et s'écrit `` [`A | `B of int] ``. Plus généralement, les expressions telles que `` [< `A | `B of int | `C of int | `D > `A | `B] `` sont autorisées, cette dernière indiquant tout type ayant au plus les constructeurs `` `A ``, `` `B ``, `` `C `` et `` `D `` cités mais au moins `` `A `` et `` `B ``...

Types objets : les types objets sont semblables aux interfaces de Java ou de C# en cela qu'ils expriment le contrat rempli par un objet, par la liste des méthodes implantées, munies de leur signature typée.

Exemple : `< get: unit -> int ; dup: 'a.'a -> ('a * 'a) >`. Les types objets imposent que les types polymorphes de méthodes soient explicitement liés par un préfixe. Ces variables de types sont traitées spécialement : les expressions de méthodes polymorphes ne peuvent être unifiées qu'avec une expression équivalente ayant des variables de types aux mêmes emplacements.

Outre cette forme de polymorphisme, les types objets expriment de plus un *polymorphisme de rangée*, car la liste de méthodes peut être ouverte, comme dans : `< to_string: unit -> string ; .. >` où (..) est appelée variable de rangée.

1.2.1.3 Valeurs et opérations

Différents mécanismes permettent de créer ou d'inspecter des valeurs associées aux types Caml.

Constructeurs de valeurs. Les types enregistrement et variant définissent explicitement les constructeurs de valeurs.

Ainsi le type `type e = {x:int; y:int}` permet de définir des valeurs comme par exemple `{x=3;y=5}` et le type variant `type v = A of int * v | B` définit des constructeurs `A` (non constant, parce qu'il a besoin d'arguments) et `B` constant : l'expression `A(3,B)` définit ainsi une valeur de type `v`.

Les valeurs des types fonctionnels sont construites à l'aide de l'abstraction fonctionnelle `fun` comme dans `fun x -> x + 1` et les n-uplets à l'aide de « `,` », l'opérateur virgule. Les variants polymorphes sont construits à la manière des variants (sauf qu'il n'y a pas de liste de constructeurs prédéfinie) et les objets à l'aide du mot-clé `new`.

Les valeurs de types abstraits ne peuvent être construites qu'à l'aide d'opérateurs ad-hoc externes au langage ou bien faisant partie du noyau Caml. C'est le cas des entiers, des flottants, des chaînes de caractères etc..., qui sont construits au moyen de littéraux...

Filtrage de motifs. Certaines valeurs, comme les valeurs fonctionnelles ou la plupart des valeurs des types abstraits, ne sont pas décomposables. Toutefois il existe en Caml un moyen puissant pour analyser les valeurs des types natifs ou algébriques : le filtrage de motifs.

Si on reprend l'exemple du type `v = A of int * v | B`, on peut filtrer une expression `e` de type `v` de la manière suivante :

```
match e with
  | B           -> e1
  | A(0,_)      -> e2
  | A(1,B)      -> e3
  | A(_,A(2,_) -> e4
  | _           -> e5
```

où `e1`,...,`e5` est l'expression évaluée selon le résultat du filtrage de `e`.

Le filtrage de motifs s'utilise sur les valeurs de types variants, enregistrements et variants polymorphes, les n-uplets, les entiers, les flottants, les caractères et chaînes de caractères, les booléens, les tableaux, les listes, et n'importe quelle imbrication de ces structures.

La puissance du filtrage réside dans la possibilité de lier des fragments du motif analysé, comme dans :

```
match e with
  | A(x,B)
  | A(0,A(x,_)) -> 2 * x
  | A(x,A(y,B)) -> x + y
  | _           -> 0
```

1.2.2 Architecture du compilateur Caml

Le compilateur Objective Caml développé par l'équipe Cristal/Gallium de l'INRIA existe sous deux versions : `ocamlc` produisant du code-octet et `ocamlopt` du code natif.

L'arbre de la figure 1.10 représente la chaîne de compilation du compilateur Objective Caml. La bifurcation correspond au choix de l'un des deux compilateurs `ocamlc` ou `ocamlopt`. Sur les nœuds figurent les modules contenant les différentes représentations intermédiaires et sur les arêtes les modules principaux participant aux transformations d'une représentation à une autre.

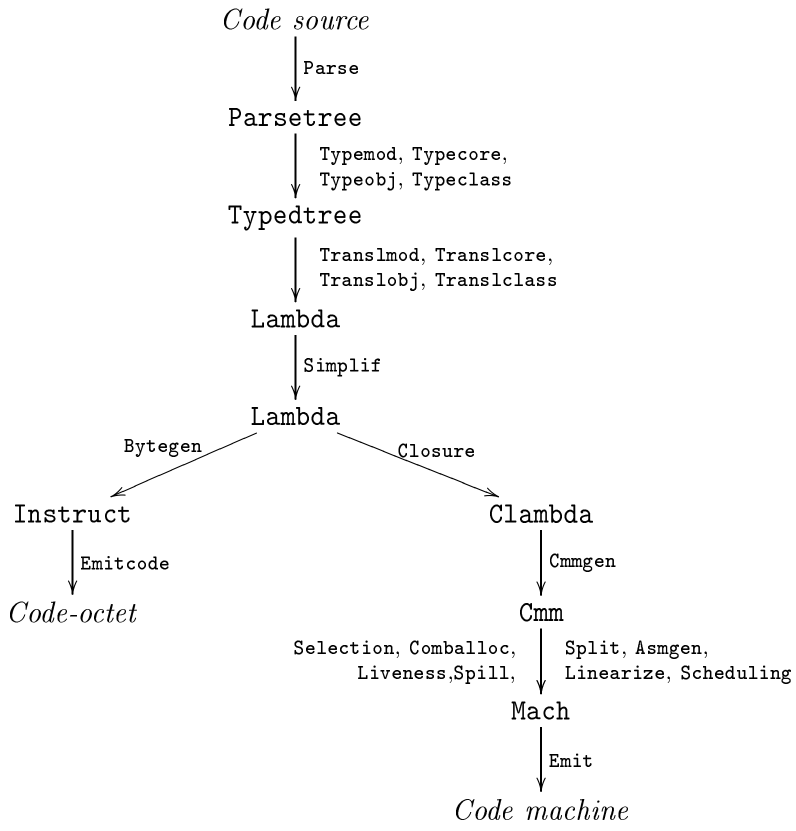


FIG. 1.10 – *Chaîne de compilation Caml*

Un fichier d'implantation est tout d'abord soumis aux phases d'analyses lexicale et syntaxique. L'arbre de syntaxe abstraite qui en ressort passe ensuite par le typage statique de Objective Caml qui lorsque le programme est bien typé retourne un arbre de syntaxe typé. Celui-ci subit ensuite différentes transformations entre des représentations intermédiaires successives jusqu'à fournir du code-octet ou du code natif. La première d'entre elles consiste à obtenir du code `Lambda`, un langage basé sur le λ-calcul mais enrichi par des opérations impératives, des structures de contrôle et différentes primitives manipulant des valeurs entières, flottantes, booléennes, des chaînes de caractères ainsi que les blocs Caml. Le code `Lambda` subit une passe de simplification à travers le module `Simplif`, optimisant entre autres l'utilisation des variables locales.

La branche du compilateur produisant du code-octet se base sur `Lambda` et engendre les instructions au format Caml `Instruct` puis sous forme physique en mémoire ou sur le disque.

Pour ce qui est de la branche du compilateur produisant du code natif, d'autres transformations sont utilisées. Le code `Lambda` est transformé en code `Clambda`, représentation intermédiaire qui gère explicitement les fermetures comme des structures de données et optimise l'application.

Les étapes suivantes prennent en charge la compilation du code Caml vers l'architecture native : le code `Cmm` (correspondant à un langage C-- manipulant explicitement les représentations internes), le code `Mach` (un langage machine fait de pseudo-instructions basées sur un processeur abstrait), et enfin le code natif (obtenu à partir de `Mach` en concrétisant les caractéristiques de l'architecture cible). Les optimisations effectuées lors de ces dernières passes concernent des problèmes classiques des back-ends (transformations finales) de compilateurs natifs, comme la politique d'attribution des registres du processeur par exemple. Enfin, le code natif pour l'architecture visée est émis sur le disque.

Dans tous les cas, il est très important de noter que si l'information de types est présente dans l'arbre de syntaxe typé à l'entrée des transformations `translcore`, `translmod`, `translclass` etc..., à la sortie les types sont abandonnés et absolument aucune information de typage n'est transmise à travers les représentations suivantes `Lambda`, `Clambda` et ainsi de suite. Le typage statique de Objective Caml garantit la sûreté à l'exécution et l'environnement d'exécution n'a besoin de gérer aucun type dynamique.

1.3 Présentation du projet OCamIL

1.3.1 Définition du projet

Microsoft met en avant l'universalité de sa plate-forme vis-à-vis des langages de programmation, concept formulé en anglais par l'expression « *language agnosticism* ». Il est vrai que les langages portés sur .NET sont désormais nombreux, ce qui semble confirmer l'éclectisme annoncé.

Il nous faut cependant évaluer la situation en ce qui concerne les langages fonctionnels, et Objective Caml en particulier.

1.3.1.1 Contexte : les langages fonctionnels sur .NET

De nombreux langages fonctionnels sont désormais portés sur .NET, mais il est important pour chacune de ces adaptations d'estimer la fidélité à l'implantation de référence, ainsi que d'évaluer la richesse de l'approche retenue en matière d'interopérabilité.

• Microsoft a favorisé le développement de deux langages dans son laboratoire de recherche à Cambridge, SML.NET et F#.

- SML.NET fait figure d'exemple. Non seulement la compatibilité avec SML est sans reproche, mais en plus l'implantation est très efficace (même si pour cela une approche de monomorphisation globale a été retenue). L'interopération est également très aboutie : en choisissant d'ajouter à SML des constructions rappelant celles de C# (notamment un modèle objet et des primitives de manipulation de types dynamiques), l'accès aux autres langages de la plate-forme est transparent. Si on peut regretter l'absence d'un toplevel, en revanche l'implantation propose une intégration à l'environnement Visual Studio.

- F# est une autre très bonne réalisation, qui a grandit parallèlement avec le travail de cette thèse. Il s'agit d'une version de Caml-Light proposant le noyau fonctionnel et impératif de Caml, mais pas le système complet de modules ni la couche object de Objective Caml. Le choix retenu en matière d'interopération est similaire à celui de SML.NET : le modèle objet de C# a été rajouté au langage Caml. Le toplevel, indisponible dans un premier temps, a été rajouté très récemment. De plus, le code produit jouit d'une efficacité raisonnable.

• Dans le sillage de ces projets, des efforts indépendants ont conduit à expérimenter la plate-forme à travers d'autres langages fonctionnels, dont les principaux sont :

- Mercury [32] est un langage logique et fonctionnel possédant plusieurs back-ends ainsi qu'une interface d'appels externe compatible avec plusieurs langages. Ce fut l'un des premiers langages fonctionnels à voir le jour sur .NET, sous l'impulsion de Microsoft (qui encouragea certains projets tiers visant à porter de nouveaux langages sur sa plate-forme). Les différents back-ends que propose l'implantation de Mercury (code-octet Mercury, C, Java...) ont été rejoints par une version émettant du code-octet CIL, mais qui reste en version beta (les bibliothèques de Mercury n'ont pas toutes été portées pour le back-end .NET). D'autre part il est possible d'incorporer aux modules Mercury des sections écrites en C#, ce qui permet virtuellement l'interopération avec n'importe quel langage .NET ; cela dit l'interface entre ce code C# et le langage Mercury lui-même n'est pas très évoluée.
- Hugs98 for .NET est un interpréteur Haskell qui permet l'accès à des composants .NET par l'ajout d'une interface d'appels de méthodes externes (il n'y a pas portage du langage). Cette approche nécessite la cohabitation de deux environnements d'exécution (l'interpréteur Haskell et la plate-forme .NET) et utilise un générateur automatique de code souche pour une utilisation haut niveau des composants .NET. On peut également citer le projet Haskell.net qui fait l'objet d'une publication [59] mais ne propose pas encore d'implantation.
- Dot-Scheme [63] repose sur un mécanisme similaire, avec deux environnements d'exécution. L'interface d'appels externes s'insère plus naturellement dans un langage aux types dynamiques comme Scheme, et s'appuie sur la bibliothèque `Reflection`.
- Bigloo [11] quant à lui franchit le pas en compilant directement les programmes Scheme vers l'environnement .NET, en plus des autres back-ends pour Java et C. La version .NET est encore considérée comme expérimentale ; très complète du point de vue des caractéristiques du langage source, le mécanisme d'interopération sera certainement analogue à ce qui est fourni pour Java (utilisation de l'API Java au moyen de clauses spéciales qui peuvent être engendrées automatiquement à partir de fichiers `.class` grâce à un outil appelé `Jigloo`).
- MoscowML for .Net [51], compile SML directement sur la plate-forme .NET. Cependant, ses possibilités d'interopérabilité se limitent pour l'instant à l'appel de méthodes statiques.
- Nemerle [60] est un langage objet et fonctionnel spécialement conçu pour la plate-forme .NET. Le modèle objets est celui du CTS et l'interopération est

permise de façon très intégrée.

Parce qu'ils se démarquent du paradigme objet très en vogue à l'heure actuelle, les langages fonctionnels se sont avérés être de très bons candidats pour tester la flexibilité de la plate-forme .NET. À la lecture des rapports d'implantations ou à l'examen des prototypes eux-mêmes, on peut constater que bien souvent, malgré la publicité, la plate-forme ne supporte pas de manière très adaptée tous les paradigmes de programmation, particulièrement la programmation fonctionnelle.

Les principales difficultés concernent :

- L'adaptation du système de types CTS au polymorphisme paramétrique.
- Les difficultés de représentation de certaines structures chères aux langages fonctionnels, comme les fermetures.
- L'inadéquation entre le modèle objet proposé par la plate-forme et celui défini par les langages. C'est le cas par exemple du modèle de classes de Objective Caml dont les caractéristiques sont discutées à la section 4.2.

1.3.1.2 Extensions de la plate-forme .NET

Le développement de projets comme SML.NET et F# fut l'occasion pour les équipes de recherche de Microsoft d'élaborer deux extensions à la plate-forme standard :

- ILX [82] incorpore des constructions destinées aux langages fonctionnels : en particulier il est proposé de définir des types fermetures et d'ajouter des instructions CIL pour les manipuler directement.
- les Generics [50] introduisent une forme intéressante de polymorphisme dans la plate-forme .NET.

L'extension ILX n'a jamais été intégrée à la plate-forme .NET. Les types et primitives introduites pour les fermetures définissent une cible commode pour les compilateurs mais ne font pas partie du cœur de la plate-forme. L'article [82] discute des traductions possibles vers le jeu d'instructions standard de CIL (ainsi que sa version Generics). Dans le domaine des valeurs fonctionnelles, Microsoft a jusqu'à présent préféré ajouter des constructions basées sur du sucre syntaxique : la version 2.0 du CLI permet l'utilisation de délégués anonymes, et la nouvelle version 3.0 introduit des lambda-expressions dans le langage C# (prises en charge dans le compilateur sans modification de l'environnement d'exécution).

En revanche, l'extension Generics a effectivement été ajoutée dans la version 2.0, sortie en octobre 2005. Cette extension permet l'utilisation de paramètres de types[9] dans les classes, interfaces, types valeurs et méthodes. La plate-forme .NET a été

9. Les types paramétrés sont invariants par rapport à leurs paramètres (c'est-à-dire que si `T` $\leq$ `U`, il n'y pas de relation entre `List<T>` et `List<U>`).

étendue afin de prendre en charge ces constructions (l'article [50] décrit en détails les principes et l'implantation des Generics).

- Le système de types CTS est enrichi afin de manipuler des déclarations, des références et des instanciations de types polymorphes. Ainsi il est possible de définir un type de liste générique `List<T>` et de manipuler des expressions de type `List<T>`, `List<int>`, `List<string>`, `List<List<U>>` etc... C'est également possible dans les types d'une signature de méthode.
- Les instructions CIL ont été étendues pour prendre en charge ces nouveaux types.
- Les instanciations de types et la génération de code spécialisé sont réalisées par l'environnement d'exécution lors du mécanisme de JIT. La plate-forme maintient à l'exécution le type instancié exact des valeurs.

Les types paramétrés introduits par les Generics offrent un cadre naturel à l'implantation du polymorphisme paramétrique défini dans le noyau des langages de la famille ML (mais ils ne se généralisent pas au polymorphisme d'ordre supérieur introduit par les systèmes de modules de SML et de Caml : plus de détails sur ce sujet à la section 3.1.4.2).

L'implantation actuelle des Generics spécialise le code sur les instanciations à des types valeurs et partage un même code entre types références. L'instanciation de paramètres de types à des types valeurs permet l'élimination des représentations encapsulées. L'idée principale des Generics est de réaliser la monomorphisation à l'exécution, ce qui garantit d'avoir exactement les versions du code utiles et permet d'avoir une véritable compilation séparée (il n'est pas nécessaire de connaître globalement à l'avance toutes les utilisations possibles du code polymorphe).

1.3.1.3 Le projet OCamIL et ses objectifs

Synthèse. L'état des lieux qui précède nous inspire ces conclusions :

- Les langages fonctionnels sont apparemment bien représentés sur la plate-forme .NET, cependant nombre d'adaptations restent très incomplètes.
- Le langage Objective Caml ne dispose pas d'implantation rigoureusement fidèle. En effet le langage F# a choisi de s'en écarter afin de mieux s'intégrer à la plate-forme.
- Il est intéressant de mener à son terme un projet d'adaptation complète du langage Objective Caml : cela permet de mesurer la pertinence de la plate-forme pour la compilation d'un langage fonctionnel ainsi que la capacité d'ouverture d'un langage comme Caml et de son implantation à un environnement d'exécution différent et récent.

À travers le développement d'un compilateur complet nous proposons une expérience pratique, permettant de tirer des conclusions concrètes de manière indépendante (en dehors des centres de recherche de Microsoft).

Définition du projet OCamIL. Le projet OCamIL vise à adapter le compilateur Objective Caml pour produire des assemblages .NET, sous la forme d'exécutables ou de bibliothèques partagées. Notre espoir est de permettre la diffusion du langage Objective Caml sur une nouvelle plate-forme, et en retour de pouvoir bénéficier des atouts de cette plate-forme. En particulier, il sera particulièrement intéressant d'interopérer avec des composants .NET et ainsi utiliser les bibliothèques conçues pour cet environnement. Nous prendrons en compte dans la conception de OCamIL la possibilité d'utiliser certains outils liés à la plate-forme : on pourra se servir de débogueurs ou de profileurs .NET, ou encore intégrer OCamIL dans des *IDE* (environnements de développement intégrés) comme Visual Studio.NET...

Voici les trois principes, par ordre décroissant d'importance, qui ont orienté les choix de conception de OCamIL:

1. **Compatibilité.** La garantie de compatibilité avec l'implantation de référence de Objective Caml, développée au sein de l'équipe Cristal/Gallium à l'INRIA est primordiale. En particulier nous imposons de ne modifier ni la syntaxe ni la sémantique du langage et surtout de ne sacrifier aucun trait de programmation présent en Objective Caml, y compris la structuration par modules et la couche objet. Nous espérons rendre aussi transparent que possible le développement d'un projet Caml sur la nouvelle plate-forme.
2. **Interopérabilité.** Nous voulons tester les capacités d'interopération offertes par la plate-forme. En particulier, interfacer Objective Caml avec le langage C# semble prometteur. La confrontation du modèle objet de Objective Caml et du CTS de la plate-forme d'exécution sera au cœur du problème de l'interopérabilité.
3. **Performances.** Enfin, il s'agit de ne pas sacrifier les performances au profit des points précédents.

Ces objectifs sont autant de contraintes, parfois antagonistes. Ainsi compatibilité et performances peuvent être difficiles à concilier. De même, conserver la sémantique du langage n'est pas sans conséquence dans un cadre d'interopération multi-langages. Nous verrons cela en détail au chapitre 4.

Nous choisissons de nous intéresser prioritairement à la **génération de code géré vérifiable** : le code géré bénéficie de tous les services de la plate-forme .NET (récupération automatique de mémoire, interopération, introspection de types, sérialisation sûre et débogage, entre autres) et le code vérifiable introduit des garanties de bonne conduite du code compilé dans un cadre multi-langages.

1.3.2 Architecture du compilateur OCamIL

Le compilateur OCamIL répond à un problème bien défini : il s'agit, dans le cadre des contraintes que nous nous sommes fixées et qui ont été énoncées dans la section précédente, de compiler un langage fonctionnel statiquement typé, dont l'environnement d'exécution est traditionnellement dépourvu de types, vers une machine à pile reposant sur un système de types orienté objets. On pourra pour cela se baser sur l'implantation de référence de Objective Caml, dont les sources sont libres.

1.3.2.1 Dispositif de compilation vers .NET

Expressions et code compilé. Dans le cadre d'un langage fonctionnel à sémantique d'appel par valeurs et d'une machine virtuelle à pile, la relation entre expressions et code compilé se schématise comme indiqué sur la figure 1.11.

Soit une expression `e = C(e1,...,en)`, formée de sous-expressions `e1` ... `en` et du constructeur `C`. L'évaluation de `e` repose sur l'évaluation préalable des sous-expressions, ce qui d'un point de vue d'une sémantique à grands pas se note sous la forme :

$$\frac{e_1 \Downarrow v_1 \ldots e_n \Downarrow v_n}{e \Downarrow v} \quad C$$

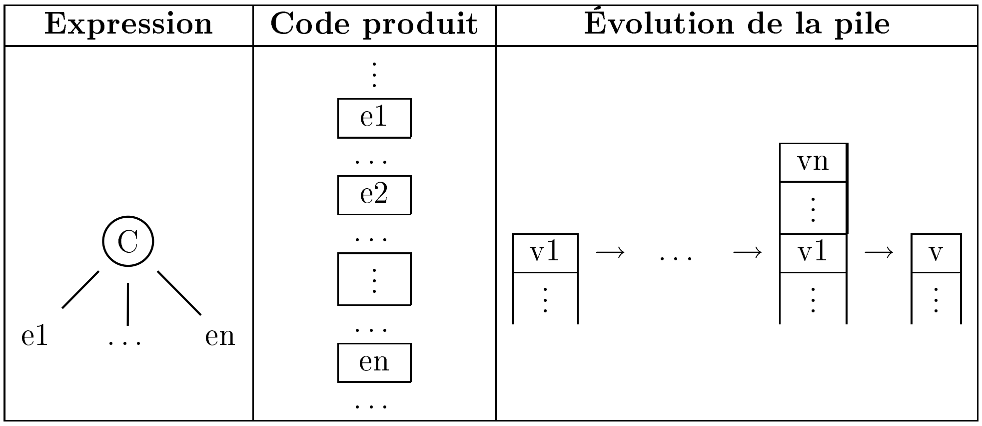
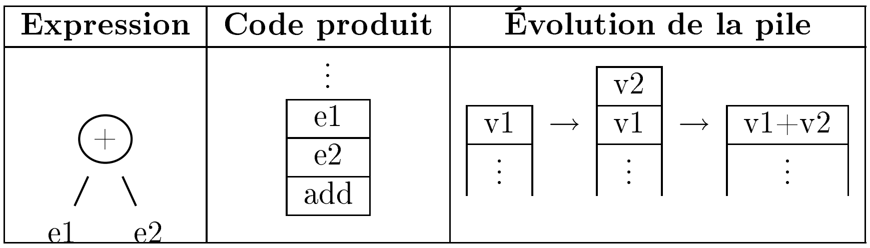


FIG. 1.11 – *Compilation d'une expression Caml*

Le code compilé pour `e` est obtenu en compilant récursivement les expressions `e1`...`en`, en considérant l'invariant suivant : **l'exécution du code compilé pour `e` laisse sur la pile une valeur de la plate-forme d'exécution qui représente `v`.**

Du point de vue de la pile, on a donc une accumulation des valeurs correspondant aux évaluations $e_1 \Downarrow v_1 \ldots e_n \Downarrow v_n$, suivie par une consommation de ces valeurs par un opérateur laissant ensuite sur la pile la valeur de `e`, réalisant donc $e \Downarrow v$.

Ainsi dans le cas simple de l'addition $\dfrac{e1 \Downarrow v1 \qquad e2 \Downarrow v2}{e1 + e2 \Downarrow v1 + v2}$, on obtient :



Toutes les sous-expressions ne sont pas forcément évaluées, et cela peut mener à utiliser des expressions de branchements conditionnels, comme dans le cas d'une instruction `if`, dont la sémantique à grands pas est :

$$\frac{eb \Downarrow true \qquad e1 \Downarrow v1}{\texttt{if}\ eb\ \texttt{then}\ e1\ \texttt{else}\ e2 \Downarrow v1} \qquad\qquad \frac{eb \Downarrow false \qquad e2 \Downarrow v2}{\texttt{if}\ eb\ \texttt{then}\ e1\ \texttt{else}\ e2 \Downarrow v2}$$

Expression	**Code produit**	**Évolution de la pile**
ifthenelse eb e1 e2	⋮ $\boxed{\text{eb}}$ brfalse F $\boxed{\text{e1}}$ br E F: $\boxed{\text{e2}}$ E: ⋮	$\boxed{\text{true}}$ ⋮ $\rightarrow$ $\boxed{\text{v1}}$ ⋮ ou $\boxed{\text{false}}$ ⋮ $\rightarrow$ $\boxed{\text{v2}}$ ⋮

Correspondance de types. Le modèle de compilation décrit ci-dessus montre la correspondance entre valeurs d'une expression et valeur sur la pile d'exécution d'une méthode CIL et par là-même entre type de l'expression et type de l'élément empilé.

En réalité, il n'y a pas que la pile qui soit concernée. Les différents emplacements mémoire qui peuvent contenir des valeurs (pile, variables locales, arguments, champs d'objets) sont typés par le CTS et il nous faut éclaircir la relation entre le type inféré pour une expression Caml donnée et le type CTS qui lui correspondra dans le code compilé.

Cela suppose tout d'abord qu'une correspondance entre les deux systèmes de types est possible, et impose ensuite au compilateur OCamIL de pouvoir modéliser les types des expressions compilées afin de générer du code correct. C'est l'objet de la section 3.1.

1.3.2.2 Structuration du compilateur

Le choix du back-end. Plutôt que de développer de toutes pièces un nouveau compilateur, nous avons décidé d'implanter le compilateur OCamIL comme un composant greffé sur le compilateur Objective Caml standard. Les sources de OCamIL forment des rustines (*patches*) à appliquer aux sources de Caml, provoquant la modification ou l'ajout de fichiers d'implantation et d'interface.

Il faut également décider à quel moment se brancher sur le compilateur Objective Caml et en particulier savoir si on utilise plutôt le compilateur de code-octet ou le compilateur natif. Le code-octet .NET ne ressemble en rien au code-octet de Caml (par exemple ce dernier est optimisé pour la gestion des fonctions) et nous avons préféré profiter de l'explicitation des fermetures réalisée au cours du passage de la représentation `Lambda` à la représentation `Clambda`, ce qui entraîne que le compilateur OCamIL dérive de la branche de compilation native de Objective Caml.

Le choix du back-end, qui détermine en grande partie l'architecture du compilateur OCamIL, procure un grand nombre d'avantages :

• **Commodité de développement.** Tout d'abord, le langage Caml est particulièrement bien adapté à l'écriture de compilateurs (la compilation est l'un des domaines de prédilection des langages fonctionnels). De plus il est par tradition son propre langage d'implantation : il reste naturel de procéder de la même manière pour développer une variante.

La commodité dégagée par cette approche est particulièrement apparente lors de la mise au point, car ce choix pousse à une logique de *bootstrap*. Cela permet de tester rapidement la conformité de la nouvelle implantation sur un projet de taille appréciable : le compilateur lui-même. Atteindre un point fixe dans le cycle de bootstrap est déjà une très bonne garantie (empirique, certes) de correction.

• **Réutilisation de code.** Écrire de toutes pièces un compilateur pour un langage comme Caml n'est pas une mince affaire. À titre d'exemple, le système de types de Objective Caml est assez complexe, et il nous semble très raisonnable de ne pas chercher à réinventer l'algorithme d'inférence de types de Objective Caml. Autre exemple, le compilateur de l'INRIA utilise de nombreuses optimisations dont OCamIL peut directement tirer partie, un certain nombre ayant lieu relativement tôt dans la chaîne de compilation.

Tirer partie de ces pré-traitements ne constitue pas uniquement un gain en terme de temps de développement, comme l'indique le point suivant.

• **Compatibilité.** Énoncé comme prioritaire, le critère de compatibilité joue largement en faveur de notre choix : toutes les passes prises en charge par le compilateur standard avant intervention du back-end OCamIL seront compatibles par construction.

De plus il faut voir la contrainte de compatibilité comme une contrainte dynamique. Objective Caml a toujours été considéré comme un laboratoire à idées dans le domaine des langages fonctionnels et sa spécification est amenée à évoluer à un rythme soutenu [10]. Si bien sûr notre approche ne peut garantir un passage automatique aux nouvelles versions, elle tend à limiter les adaptations à un niveau d'interface, sans besoin de retoucher aux phases en amont du greffon OCamIL.

• **Ouverture.** Nous ne voulons pas que le projet OCamIL soit fermé. Comme justifié précédemment, il se donne toutes les chances d'évoluer de consort avec le compilateur Caml de référence. De plus, les termes des licences de Caml et de OCamIL ainsi que le libre accès aux sources autorise chacun à apporter sa pierre à l'édifice.

Le choix de la plate-forme d'exécution. Le compilateur décrit dans ce mémoire vise la plate-forme .NET, versions 1.1 et supérieures. Nous n'exploitons pas les constructions introduites par les Generics depuis la version 2.0 de .NET, d'une part parce qu'elles ne résolvent pas le problème des représentations polymorphes pour l'ensemble du langage Objective Caml (le polymorphisme d'ordre supérieur introduit par le système de modules de Caml n'est pas transposable dans les Generics, voir l'article [50]) et d'autre part pour rester compatible avec les machines virtuelles traditionnelles (plate-forme .NET sans Generics, machines Java).

Les méthodes utilisées dans notre prototype du compilateur OCamIL et décrites

10. Le passage d'une version de Objective Caml à la suivante se fait en moyenne une fois par an.

dans ce mémoire peuvent a priori être étendues afin d'exploiter les Generics, ce qui peut constituer un prolongement naturel de ce travail de thèse.

Chaîne de compilation et représentations intermédiaires. L'architecture du compilateur OCamIL se présente comme sur la figure 1.12. Nous avons ajouté une nouvelle dérivation de la chaîne de compilation au niveau des représentations intermédiaires `Lambda` et `Clambda`.

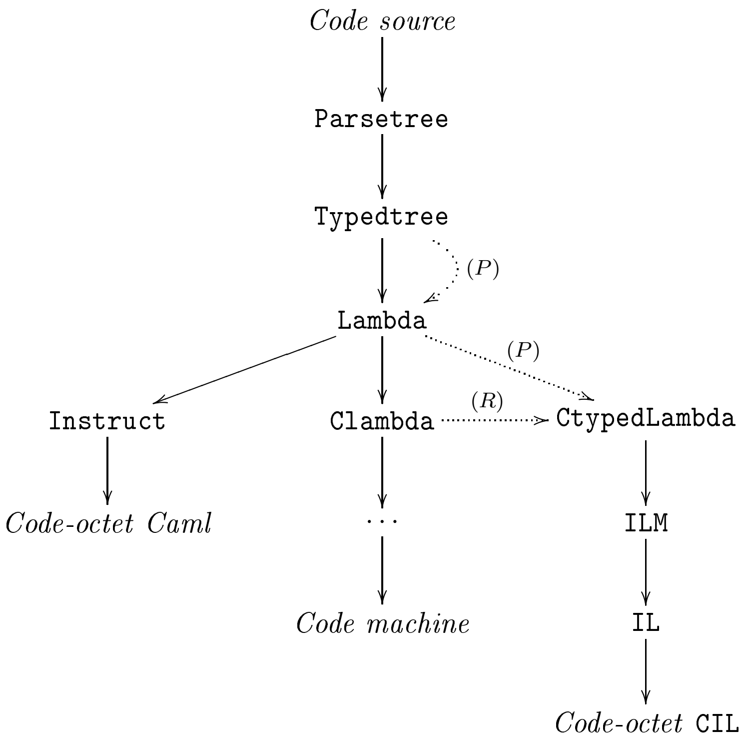


FIG. 1.12 – *Chaîne de compilation OCamIL*

Le compilateur OCamIL introduit trois nouvelles représentations intermédiaires : `Ctypedlambda`, `ILM` et `IL`.

- `Ctypedlambda` est le produit d'une décoration de la représentation intermédiaire `Clambda` au moyen d'annotations de types.
- `ILM` est un langage de macro-instructions qui s'expansent en une suite d'instructions CIL.
- `IL` décrit les instructions CIL dans une grammaire Caml. Cette dernière représentation permet l'émission directe de code.

Le détail des transformations d'un programme source Caml en ces différentes représentations est donné dans les chapitres 2 et 3.

Le problème de l'information de typage. Chacune de ces trois nouvelles représentations intermédiaires est indissociable d'une grammaire de types, dont la fonction est d'assurer un bon choix d'implantation des valeurs Caml dans le système de types de la machine cible.

Le diagramme 1.12 montre deux chemins qui aboutissent à `Ctypedlambda` : en effet nous avons expérimenté dans notre implantation deux méthodes pour obtenir l'information de types liée à `Ctypedlambda`.

– Par reconstruction de types (R) : les informations de typage sont reconstruites à l'examen de la représentation `Clambda` (en s'aidant en particulier des primitives de manipulation des valeurs).
– Par propagation de types (P) : les informations de typage sont propagées tout au long de la chaîne de compilation depuis la sortie de l'algorithme d'inférence de types de Caml.

Le détail de ces deux méthodes et la connexion avec la chaîne de compilation standard de Objective Caml en amont afin de parvenir à la première des nouvelles représentations intermédiaires `Ctypedlambda` fait l'objet des sections 2.2.2 et 2.2.3.

Chapitre 2

Environnement d'exécution et typage

Ce chapitre se penche sur les problématiques rencontrées lors de l'élaboration du front-end de OCamIL, principalement le typage des représentations intermédiaires transmises par le compilateur Caml d'origine et la manière d'adapter ces types au système CTS imposé par l'environnement d'exécution de .NET.

Sommaire

La première section donne des informations de bas-niveau sur le mécanisme de compilation par le compilateur Caml de référence (en détaillant notamment les langages intermédiaires utilisés dans la chaîne de compilation et la représentation des valeurs dans l'environnement d'exécution Caml) et examine les problèmes de typage rencontrés.

La deuxième section explique les différentes méthodes utilisées dans le compilateur OCamIL pour reconstituer des informations de types nécessaire à la compilation vers .NET.

2.1 Compilation du langage Objective Caml

2.1.1 Environnement d'exécution et représentations Objective Caml

Nous détaillons l'environnement d'exécution utilisé par le compilateur de code natif.

L'environnement d'exécution du langage Objective Caml doit tenir compte des contraintes suivantes :

- la mise en œuvre efficace du paradigme fonctionnel : implantation optimisée des fermetures, prise en charge de la récursivité terminale,
- la manipulation intensive de données structurées (types variant, types enregistrement...),
- le polymorphisme paramétrique,
- la récupération automatique de mémoire,
- la conservation d'une efficacité raisonnable sur les types de base tels que les entiers ou les flottants.

2.1.1.1 Entiers et blocs

La représentation des valeurs Caml repose sur une dichotomie fondamentale : les entiers (ainsi que les booléens et les caractères[1]) sont codés par des entiers machine et toutes les autres valeurs sont représentées en mémoire par un bloc, c'est-à-dire un pointeur vers une structure allouée dans le tas. Celle-ci est composée d'un nombre déterminé d'éléments pouvant à leur tour contenir un entier ou un pointeur sur un bloc.

Cette représentation uniforme est possible grâce à un bit de tag (une ruse déjà utilisée par le compilateur Smalltalk-80 [42]). L'implantation de Objective Caml code dans les entiers machine à la fois les entiers et les blocs Caml de la manière suivante : partant de la constatation que les blocs sont des pointeurs en mémoire alignés sur des mots et sont donc des entiers machine pairs (dont le bit de poids faible est 0), il alors est possible de représenter les entiers Objective Caml par des entiers machine impairs obtenus par un décalage logique vers la gauche et la mise à 1 du bit de poids faible. Ainsi les entiers 0,1,2,3... sont représentés par les entiers machine 1,3,5,7... (bien sûr on perd un bit d'information pour les entiers qui ont

1. Le typage statique évite de confondre entiers, booléens et caractères même s'ils ont la même représentation. Notons au passage que les caractères Caml sont sur 8 bits.

donc une taille non standard).

L'intérêt est de pouvoir représenter de manière uniforme les entiers et les blocs, qui peuvent ainsi figurer dans les mêmes emplacements mémoire, et se comporter de manière transparente vis à vis du polymorphisme. Cela a pour conséquence d'éviter de « boxer » (emboîter) les entiers et ainsi d'éviter des indirections supplémentaires qui peuvent être sources de lourdes inefficacités. De plus, le récupérateur automatique de mémoire pourra discriminer les blocs (relogeables) des entiers en se basant sur la valeur du bit de poids faible.

Le codage un peu particulier des entiers a deux conséquences principales :

- Comme signalé plus haut la place prise par le bit informatif est perdu pour la valeur entière elle-même. Ainsi les entiers Objective Caml sont des entiers 31 bits sur une architecture 32 bits et 63 bits sur une architecture 64 bits. Il existe par ailleurs des types spéciaux `int32`, `int64` et `native int` qui ont bien la taille indiquée, mais qui sont représentés par des blocs encapsulant la valeur correspondante : leur utilisation est bien plus coûteuse que les « entiers Caml ».
- Il est nécessaire d'adapter les opérations arithmétiques : ainsi l'addition de deux entiers x et y est compilée comme $x+y-1$, de façon à ne pas ajouter les deux bits de poids faible mis à 1 ($+$ et $-$ désignent les opérations processeur). De même la multiplication est $(x-1)*(y>>1)+1$. Le supplément d'opérations a toutefois un coût négligeable comparé aux indirections que l'on évite en contrepartie.

2.1.1.2 Les blocs en détail

Structure des blocs. Un bloc est formé d'un premier mot d'en-tête, suivi des données. L'en-tête contient la taille du bloc, son type, et réserve 2 bits de « couleur » au ramasse-miettes à des fins de marquage, ce qui, sur une architecture 32 bits[2] donne un mot d'en-tête de la structure suivante :

Bits	31 ... 10	9 ... 8	7 ... 0
Usage	Taille en mots	Couleur	Tag

Le tag, exprimé sur 8 bits, est disponible pour la représentation des types utilisateurs, à l'exception de quelques-uns réservés par l'implantation Caml et listés sur la figure 2.1.

Les tags « fermeture » et « fermeture infixe » sont utilisés par les valeurs fonctionnelles, que nous étudions en détail un peu plus bas. D'autres types de valeurs sont identifiables par un tag réservé : chaînes de caractères, flottants, tableau de flottants, blocs abstraits, ce qui permet de traiter ces valeurs de manière particulière, en effet :

- Les flottants et les blocs abstraits, bien qu'étant des blocs, doivent échapper au contrôle du ramasse-miettes. La différence se fait grâce au tag. Les blocs

2. Sur une architecture 64 bits, la taille du bloc en mots s'étend du bit 10 au bit 63.

Tag réservé	Usage
0xF6	bloc « lazy »
0xF7	fermeture
0xF8	objet
0xF9	fermeture infixe
0xFA	bloc « lazy forward »
0xFB	bloc abstrait
0xFC	chaîne de caractères
0xFD	flottant
0xFE	tableau de flottants
0xFF	bloc spécial

FIG. 2.1 – *Les tags réservés par l'environnement Caml*

spéciaux sont abstraits du point de vue du GC mais ont de particulier de posséder un jeu de fonctions propres pour la finalisation, la comparaison, le hash... Les constructions paresseuses sont elles aussi gérées de manière particulière, et l'environnement d'exécution réserve deux tags pour elles : 0xF6 pour les calculs non effectués et 0xFA pour les résultats de calculs paresseux.

- Les tableaux de flottants ont une représentation spéciale (les flottants composant le tableau devraient normalement faire l'objet d'une indirection puisque les flottants sont des blocs, mais par souci d'efficacité ils sont regroupés dans un bloc agrégeant les données flottantes côte à côte), si bien que des primitives génériques d'accès à un tableau doivent différencier tableaux normaux et tableaux flottants, ce qui est permis grâce au tag réservé (car tous les autres tableaux ont pour tag 0x00).
- Les chaînes de caractères ont une représentation compacte groupant plusieurs caractères 8 bits dans chaque mot du bloc (par paquets de 4 caractères pour une architecture 32 bits). Le tag réservé d'un bloc chaîne permet de le considérer comme un ensemble indivisible et soustraire ses éléments à l'habituelle dichotomie entre entiers et pointeurs.

Les tags de 0x00 à 0xF5 sont des tags utilisateurs ; voyons quel usage il en est fait suivant les types Caml :

• Les n-uplets, les tableaux (sauf tableaux de flottants) et les enregistrements ont tous un tag 0x00. Le tag ne sert pour ainsi dire à rien pour ces valeurs, et ne remplit pas de fonction discriminatrice pour le filtrage de motifs. Les structures représentant les objets Objective Caml ont aussi le tag 0x00.

• Les variants ont une implantation mixte à base d'entiers et de blocs : les constructeurs constants d'un type variant sont représentés par des entiers, et les constructeurs non constants sont représentés par des blocs possédant un tag propre, les paramètres du constructeur figurant dans les éléments du bloc.

Par exemple, si on définit : `type t = A | C of int | B | D of float`, alors les valeurs `A` et `B` sont représentées par les entiers 0 et 1, alors que `C 15` est représenté par un bloc de tag 0 contenant l'entier Caml 15 et `D 0.33` est représenté par un bloc de tag 1 contenant un pointeur vers un bloc flottant. Il y a ainsi 246 constructeurs non constants possibles dans un variant Caml (de tag maximum 0xF5).

Le filtrage de motifs sur des variants exploite la dichotomie entiers/blocs d'une part et se base sur les tags pour discriminer les blocs d'autre part. Remarquons enfin qu'il n'y a aucun risque de méprendre des valeurs d'un type variant avec des valeurs d'un autre type variant, d'autres blocs ou même de véritables entiers : c'est une garantie offerte par le typage statique.

• Les variants polymorphes ont une représentation différente : ils ont le tag 0x00 et les constructeurs sont distingués par un champ dans le bloc qui contient une valeur de hachage du nom du constructeur. En effet il est impossible d'associer une énumération entière univoque aux constructeurs car ceux-ci ne sont pas fixés une fois pour toutes dans une définition de types comme dans le cas des variants simples.

Le cas des fermetures. Une fermeture est implantée comme un bloc, qui outre son en-tête, est composé de deux parties :

- Une première section contenant le ou les pointeur(s) de code des valeurs fonctionnelles.
- Une deuxième section, appelée « environnement », qui capture les valeurs des variables libres du corps de la fonction. Cette partie peut également servir à stocker les arguments déjà renseignés dans le cas d'une application partielle.

Schématiquement, un bloc fonctionnel a donc l'allure suivante :

En-tête de fermeture	Partie fonctionnelle	Environnement

Comme nous allons le voir plus loin, la partie fonctionnelle peut contenir plusieurs fonctions dans le cas d'une définition de fonctions mutuellement récursives.

<u>Cas des fonctions non-mutuellement récursives.</u> Plaçons-nous dans le cas d'une fonction seule, avec environnement.

La partie fonctionnelle a une taille de 2 ou 3 mots, selon la valeur de *l'arité* de la fonction. Celle-ci est définie de la manière suivante :

- Les fonctions sous forme curryfiée ont une arité positive, égale au nombre de leurs arguments. De plus les fonctions ayant plusieurs arguments, dont un au moins est un n-uplet, sont vues comme des fonctions curryfiées (en considérant chaque n-uplet comme un paramètre simple) et rentrent dans cette convention de signe.
- Les fonctions totalement décurryfiées, c'est-à-dire prenant pour unique argument un n-uplet, ont une arité négative qui est égale à l'opposé de la taille du n-uplet.

Le code compilé pour le corps de la fonction accède à un nombre d'arguments égal à la valeur absolue de l'arité, auquel s'ajoute éventuellement un argument supplémentaire qui reçoit la fermeture courante. En effet, dans la représentation des valeurs fonctionnelles par fermetures, le code des fonctions est alloué statiquement tandis que les fermetures sont construites dynamiquement. Pour une même fonction, différentes fermetures peuvent être construites lors de l'exécution d'un programme, et comme le code de la fonction est unique, il est nécessaire au compilateur d'ajouter à toute fonction faisant usage de son environnement un argument conventionnel, destiné à recevoir un pointeur vers le bloc de la fermeture. Le code fait usage de ce pointeur pour accéder à l'environnement, mais aussi dans le cas de fonctions mutuellement récursives pour effectuer un appel aux autres fonctions (cas que nous détaillons ultérieurement).

Revenons à la description des blocs fermetures. La partie fonctionnelle renferme toujours au moins le pointeur de code et l'arité de la fonction. Pour une arité différente de 1, la partie fonctionnelle contient également un pointeur vers une fonction générique qui soit gère l'application partielle, soit s'occupe d'extraire les arguments d'une fonction totalement décurryfiée. Ainsi la partie fonctionnelle a l'une des trois formes suivantes :

Pointeur de code	Arité $a = 1$	fonction d'arité 1

Pointeur de code sur $\texttt{Gcurry}_{\texttt{a,0}}$	Arité $a > 1$	Pointeur de code	fonction pouvant être appliquée partiellement

Pointeur de code sur $\texttt{Gtuple}_{-\texttt{a}}$	Arité $a < 0$	Pointeur de code	fonction totalement décurryfiée

L'environnement d'exécution Caml définit les familles de fonctions génériques $\texttt{Gtuple}_{\texttt{n}}$ ($n > 1$) et $\texttt{Gcurry}_{\texttt{n,p}}$ ($n > 1$ et $0 \leq p < n$).

- $\texttt{Gtuple}_{\texttt{n}}$ a pour arité 1 mais utilise l'argument additionnel pointant sur la fermeture. Elle extrait les n champs de son argument et appelle ensuite la fonction réelle via le pointeur de code en troisième position de la fermeture.
- Les fonctions $\texttt{Gcurry}_{\texttt{n,0}} \ldots \texttt{Gcurry}_{\texttt{n,n}-1}$ ont toutes pour arité 1 et utilisent l'argument additionnel pointant sur la fermeture. Le but de $\texttt{Gcurry}_{\texttt{n,p}}$ est de stocker le p-ième argument (si on les numérote à partir de 0) dans une nouvelle fermeture. Leurs définitions sont liées de la manière suivante :
  - $\texttt{Gcurry}_{\texttt{n,0}}$ crée une fermeture du premier type ci-dessus (fonction d'arité 1), pointant sur le code de $\texttt{Gcurry}_{\texttt{n,1}}$ et possédant deux champs dans son environnement, stockant l'argument et la fermeture de départ,
  - $\texttt{Gcurry}_{\texttt{n,1}}$ fait de même, les deux champs stockés étant le nouvel argument et la fermeture précédente,
  - $\texttt{Gcurry}_{\texttt{n,2}} \ldots \texttt{Gcurry}_{\texttt{n,n}-2}$ prolongent ce schéma, contruisant une liste chaînée de fermetures stockant à chaque fois le nouvel argument,
  - $\texttt{Gcurry}_{\texttt{n,n}-1}$ prend le dernier argument et déroule la liste chaînée pour récupérer tous les arguments intermédiaires jusqu'à parvenir à la fermeture

initiale et appliquer ces arguments au code réel de la fonction.

Il reste maintenant à définir le mécanisme d'application générique[3] de Caml :

- Lorsqu'une fermeture est appliquée à un seul argument, le premier pointeur de code est appelé avec cet argument. Selon le type de fermeture, cela réalise une application totale pour une fonction d'arité 1, une application partielle à un argument (dans le deuxième type de fermeture ou celles engendrées par les appels aux fonctions `Gcurry`$_{n,p}$), ou bien l'application totale d'une fonction totalement décurryfiée via l'appel à une fonction `Gtuple`$_{n}$.
- Lorsqu'une fermeture est appliquée à $n > 1$ arguments, alors si n est égal à l'arité inscrite dans la fermeture, on appelle directement le pointeur de la fonction réelle en troisième position sur tous ces arguments. Dans le cas contraire les arguments sont pris un à un et on leur applique la fonction générique située en première position de la fermeture de départ et des fermetures résultant de ces applications. Cela gère du même coup les applications partielles et les sur-applications[4].

À noter que l'argument supplémentaire pointant sur la fermeture elle-même est toujours transmis lors des appels, chaque fonction étant libre de s'en servir ou non.

Remarquons que les fermetures sont créées à deux occasions : lors de l'évaluation d'une valeur fonctionnelle `fun` et lors d'une application partielle de fonction. Nous désignerons le premier cas par « fermeture explicite » (on peut dire aussi « syntaxique »).

<u>Exemples.</u> La table suivante donne l'allure des fermetures correspondant à quatre définitions de fonctions (ayant `a` comme variable libre).

Expression	**Bloc fermeture**				
`let f x = x + a`		$*$f	1	a	
`let g x y z = x + y + z + a`		$*$`Gcurry`$_{3,0}$	3	$*$g	a
`let h (x,y) = x + y + a`		$*$`Gtuple`$_{2}$	-2	$*$h	a
`let k (x,y) z = x + y + z + a`		$*$`Gcurry`$_{2,0}$	2	$*$k	a

3. Il existe aussi une application directe : pour celle-ci le code appelle directement le pointeur de la fonction avec tous ses arguments (dont le dernier peut être une fermeture capturant les variables libres).

4. La sur-application intervient lorsqu'une fonction est appliquée à un nombre d'arguments supérieur à son arité : c'est possible lorsque cette fonction retourne une fonction qui prendra en charge à son tour les arguments additionnels.

Cas 1) L'appel de `f` sur son unique argument mène à un appel direct de son pointeur de code ∗`f`.

Cas 2) Si `g` est appliquée totalement, la fonction générique n'est pas utilisée : le pointeur de code ∗`g` est appelé directement avec tous les arguments. En revanche si l'application est partielle, c'est le pointeur $\texttt{Gcurry}_{3,0}$ qui est suivi : le code de cette fonction générique crée une nouvelle fermeture qui capture dans son environnement la fermeture précédente et l'argument de l'application partielle, comme indiqué sur le schéma ci-dessous.

	$\texttt{Gcurry}_{3,1}$	1	$arg1$	↓

	$\texttt{Gcurry}_{3,0}$	3	∗g	a

Par la suite, l'arité de la fonction générique valant 1, l'application des deux arguments restants passe toujours par la création des fermetures intermédiaires.

Cas 3) L'application de `h` à son argument n-uplet passe par la fonction générique $\texttt{Gtuple}_2$, qui décompose l'argument et appelle le code de la fonction `h` qui attend des arguments dissociés.

Cas 4) Les fonctions comme `k` sont considérées comme étant sous forme curryfiée et sont traitées comme dans le cas 2. Les arguments n-uplets sont transmis en blocs (c'est au code de la fonction de les déstructurer).

<u>Cas des fonctions mutuellement récursives.</u> Les fonctions mutuellement récursives se voient allouer une fermeture partagée. Le bloc est formé de la succession des blocs fonctionnels de chaque fonction suivi par une partie environnement commune. La structure est la suivante :

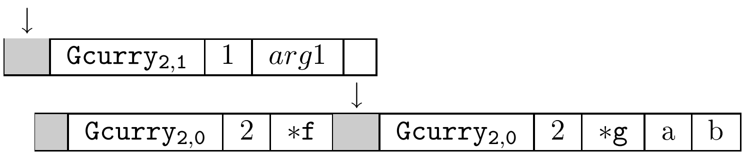
En-tête de fermeture	Bloc Fonction 1	En-tête de fermeture infixe	Bloc Fonction 2	...	Environnement partagé

Les blocs fonctionnels suivent la description donnée plus haut pour les fonctions simples, occupant 2 ou 3 mots chacun. De plus ils sont précédés d'un en-tête de bloc : Caml a recours à l'en-tête de tag spécial « fermeture infixe » pour les en-têtes internes (voir figure 2.1).

Par exemple, la définition de deux fonctions mutuellement récursives `f` et `g` d'arité 2, ayant chacune une variable libre, respectivement `a` et `b`, aboutit lors de l'exécution à la création de la fermeture suivante (les variables `f` et `g` pointant aux endroits indiqués par les flèches) :

↓				↓					
	$\texttt{Gcurry}_{2,0}$	2	∗f		$\texttt{Gcurry}_{2,0}$	2	∗g	a	b

En cas d'application partielle de `g` à un argument $arg1$, on obtient la fermeture :



Lorsque l'application sera totale et que le code de `g` est effectivement exécuté, celui-ci reçoit ses deux arguments ainsi que l'argument conventionnel supplémentaire pointant sur la fermeture partagée (au niveau du bloc infixe). Le code peut ainsi utiliser ses arguments et aussi appeler récursivement la fonction `f` en passant par la fermeture partagée.

2.1.1.3 Relations entre types et représentations

L'environnement d'exécution du langage Objective Caml est essentiellement un environnement non typé. L'inspection des valeurs permet parfois de récupérer une information sur les types Caml associés aux valeurs, mais d'une manière très incomplète. L'information se situe en premier lieu au niveau du bit de poids faible de l'entier machine représentant la valeur : s'il est nul il s'agit d'un bloc, s'il est mis à 1 c'est un entier ou une valeur d'un type variant associée à un constructeur constant. Dans le cas d'un bloc, le tag peut indiquer s'il s'agit d'une fermeture, d'une chaîne de caractères, d'un flottant ou d'un tableau de flottants mais dans tous les autres cas, il peut s'agir d'un tableau, d'un n-uplet, d'un enregistrement, d'un type variant etc.... Dans le cas d'un type variant le tag renseigne sur le numéro du constructeur mais pas sur le type variant lui-même.

De plus la relation entre types et valeurs peut être brisée par l'emploi de moyens non conventionnels. Le module `Obj` faisant partie de la bibliothèque standard (mais non documenté et dont l'utilisation est découragée) permet une manipulation de bas niveau des valeurs Caml dans un cadre non typé. En particulier la fonction `Obj.magic: 'a -> 'b` permet de casser le système de types de Objective Caml en changeant artificiellement le type d'une valeur, ce qui est parfois possible (l'uniformité de la représentation des valeurs Objective Caml permet effectivement d'envisager une valeur sous un autre type que le sien sans effectuer de copie). L'utilisation de ce module retire la garantie de sûreté d'exécution et doit rester parcimonieuse. Ce problème ne doit pas être éludé car le code du compilateur Caml lui-même a quelquefois recours à ce module.

2.1.1.4 Le langage intermédiaire `Clambda`

Le code intermédiaire `Clambda` joue un rôle clef dans l'architecture du compilateur OCamIL. C'est en effet le langage qui rend explicite les manipulations des valeurs Caml telles qu'elles sont représentées dans l'environnement d'exécution natif, en particulier la gestion des opérations de nature fonctionnelle (abstraction, application) au moyen de fermetures.

Voici sa définition telle qu'on la trouve dans les sources du compilateur Caml. Nous détaillons ensuite les différentes expressions de Clambda.

```
type ulambda =
    Uvar of Ident.t
  | Uconst of structured_constant
  | 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
  | Ulet of Ident.t * ulambda * ulambda
  | Uletrec of (Ident.t * ulambda) list * ulambda
  | Uprim of primitive * ulambda list
  | Uswitch of ulambda * ulambda_switch
  | Ustaticfail of int * ulambda list
  | Ucatch of int * Ident.t list * ulambda * ulambda
  | Utrywith of ulambda * Ident.t * ulambda
  | Uifthenelse of ulambda * ulambda * ulambda
  | Usequence of ulambda * ulambda
  | Uwhile of ulambda * ulambda
  | Ufor of Ident.t * ulambda * ulambda * direction_flag * ulambda
  | Uassign of Ident.t * ulambda
  | Usend of ulambda * ulambda * ulambda list

and ulambda_switch =
  { us_index_consts: int array;
    us_actions_consts : ulambda array;
    us_index_blocks: int array;
    us_actions_blocks: ulambda array}
```

Constantes et variables.

• `Uconst` est utilisé pour insérer une valeur constante (cela peut être un type de base ou un bloc constant).

• `Ulet(x,t1,t2)` lie la variable `x` au terme `t1` dans `t2`.

• `Uletrec` est utilisé pour définir une expression récursive ou des expressions mutuellement récursives. Par exemple `Uletrec([(x1,t1);(x2,t2)],t)` lie `x1` (resp. `x2`) à `t1` (resp. `t2`) simultanément dans `t1`, `t2` et `t`.

• `Uvar x` désigne une occurrence de la variable `x`.

Remarquons que l'analyseur syntaxique de Objective Caml associe un identificateur unique (au sein d'un fichier d'implantation) aux variables liées en fonction de leur lieur, ce qui permet de ne pas se soucier de problème de portée de variable ou d'α-conversion par la suite.

Manipulation de fermetures.

• `Ugeneric_apply(t,t1,...,tn)` est l'application du terme `t` à n arguments (notés ici `t1`,...,`tn`).

• `Udirect_apply(flbl,t1,...,tn)` est une application directe, où la fonction est connue statiquement et référencée par une étiquette (qui sera ensuite remplacée par un pointeur de code lors de l'émission de code).

• `Uclosure` définit une fermeture (une fermeture partagée dans le cas de fonctions mutuellement récursives). Ainsi `Uclosure([l1,2,[x1;x2],t],[e1;...;ep])` correspond à la fermeture d'une fonction d'arité 2, de paramètres formels `x1`,`x2`, de corps `t` et d'environnement `e1;...;ep`. L'instruction définit aussi une étiquette `l1` pour cette fermeture. Lorsque la première liste contient plusieurs définitions de fonctions, on obtient une fermeture partagée. Remarquons que l'arité peut être munie d'un signe afin de coder la curryfication, comme indiqué dans la section 2.1.1.2.

• Enfin, `Uoffset` permet d'accéder aux différentes composantes d'une fermeture, qu'il s'agisse des valeurs de l'environnement ou des pointeurs de fonctions. Dans `Uoffset(t,i)`, `t` désigne un terme qui s'évalue vers un pointeur de fonction au sein d'une fermeture et `i` est un entier relatif qui exprime le décalage dans l'environnement entre ce pointeur et le champ visé.

Boucles et exceptions.

• Les deux instructions de boucle `Uwhile` et `Ufor` ne nécessitent pas de commentaire particulier.

• `Utrywith (t1,e,t2)` évalue le terme `t1` et en cas d'exception ex-filtrante passe à l'évaluation de `t2` dans lequel l'identificateur `e` est lié à la valeur de l'exception.

Remarquons qu'il n'y a pas d'instruction `Uraise` : lever une exception se fait au moyen d'une primitive (voir plus loin).

Contrôle lié au filtrage de motifs.

• `Uifthenelse(t,t1,t2)` est l'instruction conditionnelle classique. Elle permet d'implanter l'expression `if` de Caml mais entre aussi en jeu dans la compilation du filtrage de motifs. Remarquons que le paramètre `t` contenant la condition peut être n'importe quelle valeur Caml (pas seulement des entiers Caml) : elle est utilisée pour discriminer les listes vides des listes non vides par exemple.

• `Uswitch(t,sw)` est une instruction de branchement riche, au cœur du mécanisme de filtrage de motifs. L'argument `t` est le terme à filtrer, et la structure `sw` énumère les différentes branches possibles selon que `t` est un entier Caml (de telle ou telle valeur) ou bien un bloc Caml (de tel ou tel *tag*).

• La compilation du filtrage de motifs est optimisée à l'aide des deux instructions `Ucatch` et `Ustaticfail`, qui permettent de faire sauter le flot d'exécution d'un point à un autre :

- `Ustaticfail(i,[t1;...;tn])` déclenche un saut vers un récupérateur (une cible) étiqueté par l'entier `i`. Les termes `t1`,...,`tn` sont évalués et les valeurs transmises au récupérateur.
- `Ucatch(i,[x1;...;xn],t1,t2)` est un récupérateur d'étiquette `i`. L'instruction tente d'évaluer le terme `t1` mais si un saut `Ustaticfail` de même identificateur est rencontré, alors `t2` est évalué, avec les variables `x1`,...,`xn` liées aux valeurs transmises par `Ustaticfail`.

Les identificateurs entiers permettent une imbrication riche de ces structures de contrôle. On trouvera les détails de l'utilisation de ces instructions dans l'article [35] présentant la compilation du filtrage de motifs dans l'implantation de Caml.

Divers.

• `Uassign(x,t)` est une instruction impérative, elle remplace le contenu de la variable `x` par la valeur de `t`. C'est l'implantation directe de l'expression Caml `x := t`. Remarquons que `x` référence un bloc contenant une indirection vers la valeur. Le même résultat peut être obtenu avec des primitives d'accès aux blocs (voir les primitives ci-après).

• `Usend(tm,to,t1,...,tn)` est utilisée pour l'invocation de méthodes. `tm` s'évalue vers une structure représentant une méthode, `to` vers une structure représentant une instance de classe, et `t1`,...,`tn` sont les arguments.

• `Usequence(t1,t2)` évalue séquentiellement les deux expressions, en ignorant la valeur de la première.

Primitives. `Uprim (p,t1,...,tn)` applique la primitive `p` à ses arguments[5]. En outre, chaque primitive `p` peut être paramétrée par une ou plusieurs valeurs de différentes natures.

Il faut bien distinguer ces paramètres, qui font partie intégrante de chaque occurrence de primitive et qui sont fixés statiquement, des arguments de la primitive passés par l'intermédiaire de l'instruction `Uprim` : ces derniers peuvent varier à l'exécution (ce sont des termes et peuvent être le résultat d'une évaluation).

Le typage statique de Caml garantit que les primitives reçoivent à l'exécution le bon nombre d'arguments. Les différentes primitives sont détaillées en annexe. Nous évoquons ici les plus importantes d'entre elles :

- Opérations arithmétiques : les primitives `Paddint`, `Psubint`, `Pmulint`, `Pdivint`

5. Les primitives sont définies au niveau de `Lambda` ; les deux représentations `Lambda` et `Clambda` ne diffèrent pas au niveau des primitives.

attendent deux arguments entiers et réalisent respectivement l'addition, la soustraction, la multiplication et la division. Les primitives analogues pour les flottants sont `Paddfloat`, `Psubfloat`, `Pmulfloat` et `Pdivfloat`.

- Comparaisons : la primitive `Pintcomp` prend en paramètre une comparaison (pouvant être $=$, $\neq$, $<$, $\leq$, $>$ ou $\geq$) et compare deux arguments entiers. La primitive `Pfloatcomp` fait de même pour les flottants.
- Manipulation des blocs : les deux primitives les plus importantes dans ce domaine sont `Pmakeblock` et `Pfield`. `Pmakeblock` prend en paramètre en entier *tag* et un nombre n arbitraire d'arguments : elle construit un bloc de taille n du tag spécifié et remplit les cases du bloc avec ses arguments. `Pfield` a un entier i comme paramètre et admet un argument : celui-ci doit s'évaluer vers un bloc et la primitive réalise un accès en lecture dans ce bloc à la position i.
- Appel de code C : `Pccall` prend en paramètre la description d'une fonction C et en arguments les valeurs à lui passer. Ce mécanisme est utilisé pour appeler des primitives C de l'environnement d'exécution de Caml (la comparaison polymorphe par exemple) mais aussi des composants externes (déclaration `external`).

Primitives et types.

Les opérations menées sur les valeurs Caml sont ultimement le fait des primitives, listées dans l'annexe. En l'absence d'indications de types dans le code `Clambda`, les primitives peuvent *parfois* fournir des informations sur le type des valeurs.

- Certaines primitives sont en réalité fortement typées, comme `Pstringrefs` ou `Paddint` qui ne manipule respectivement que des chaînes de caractères et des entiers.
- D'autres sont plus souples vis à vis des types du langage source. Ainsi les primitives d'accès à des blocs comme `Pfield` ne fournissent aucun renseignement sur l'origine de la valeur représentée par le bloc, fut-elle un enregistrement, un n-uplet ou un variant...

Lien avec `Lambda`.

La représentation `Lambda` précédant `Clambda` dans la chaîne de compilation de Caml s'apparente à un λ-calcul (possédant en plus des structures de contrôle et un large éventail de primitives).

La principale différence avec `Clambda` se situe au niveau de la définition et de l'application des fonctions :

- Il n'y a pas de distinction entre applications générique et directe dans `Lambda` : elles sont toutes génériques.
- Il n'y a pas de construction `closure` ni d'opération `offset` dans `Lambda` : au lieu de cela on a une définition de fonction `fun x1 ... xn` $\rightarrow$ `t` où le terme `t` peut contenir des variables libres.

2.1.2 Questions de typage

L'implantation de Objective Caml démontre la pertinence du credo « *typed programs cannot go wrong* » cher aux langages statiquement typés en évitant toute annotation des représentations mémoire par des types. Dans le cadre du projet OCamIL il nous faut toutefois tenir compte des contraintes de types imposées par la plateforme .NET. Notre problème est de combler le fossé entre un langage intermédiaire non typé et un environnement d'exécution typé, avec pour objectif :

- initialement, produire du code correct,
- ensuite, produire du code optimisé.

Prenons l'exemple suivant de génération de code CIL, qui est incorrecte pour ne pas avoir tenu compte des types mis en jeu (la variable `t` fait référence à un tableau) :

Code Objective Caml	CIL	Notes
`t.(0) + 1`	`ldloc t`	La variable locale `t` est placée sur la pile
	`ldc.i4.0`	L'entier 0 est mis sur la pile
Clambda code	`ldelem.ref`	Un élément du tableau, de type référence, est empilé
`(+ (field 0 t) 1)`	(*)	
	`ldc.i4.1`	L'entier 1 est mis sur la pile
	`add`	Addition

Au niveau de la ligne marquée par (*), le sommet de la pile contient une référence à un objet, alors que la primitive d'addition `add` demande ici une valeur entière.

Un algorithme de génération de code CIL capable de tenir compte des types aurait inséré une instruction `unbox` au niveau de (*).

Il y a donc un réel besoin de récupérer une information de types. Le typage statique conserve un rôle prépondérant pour OCamIL : s'il n'est pas le garant de la sûreté de l'exécution de manière aussi centrale que dans une implantation native de Caml (car un environnement géré peut toujours récupérer l'échec d'un programme de manière sûre), il permet d'assurer la cohérence des types CTS utilisés et donc la bonne exécution du programme.

Le système de types de Objective Caml diffère du modèle CTS inspiré par les langages objets à la Java. Voyons quels sont les traits les plus difficiles à implanter.

2.1.2.1 Définitions et expressions de types en Caml et dans le CTS

Expressivité des types en Caml et dans le CTS. En terme de typage, les paradigmes objet (à la Java/C#) et fonctionnel statiquement typé développent des points de vue différents. Alors que le premier met l'accent sur l'encapsulation et la réutilisation par héritage, le second mise sur la richesse expressive des types (types algébriques paramétrés). Dans les deux cas, les types expriment à la fois des possibilités et des restrictions. Le système de types définit un langage qui exprime le périmètre d'interaction des valeurs, les opérations qu'elles peuvent (ou ne peuvent pas) effectuer ou subir.

Cependant le modèle de classes proposé par le CTS diffère du langage de types de Caml par un emploi plus généralisé de types nommés, qui s'inscrivent dans une relation d'héritage confondue avec la notion de sous-typage (classes et interfaces nommées sont des facteurs de restriction au sens où deux types proposant le même ensemble d'opérations mais sans lien dans la relation d'héritage/implantation d'interface ne sont pas interchangeables). Au contraire les langages à la ML peuvent choisir parmi une large gamme de types algébriques, nommés ou anonymes (ceux-ci expriment leurs capacités de manière intrinsèque).

Comparons par exemple différentes manières d'implanter le point de coordonnées cartésiennes en C# et en Objective Caml.

Quelques implantations du point en C#	
Déclaration	**Utilisation**

class point {
   public int x;
   public int y;
   public void point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
	point p;
	p = new point(1,2);
	p.x = 2 * p.y;
	...

class point {
   private int x;
   private int y;
   public void point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public int X {
      get {return x;}
      set {x = value;}
   }
   public int Y {
      get {return y;}
      set {y = value;}
   }
}
	point p;
	p = new point(1,2);
	p.X = 2 * p.Y;
	...

struct point {
   public int x;
   public int y;
   public void point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
	point p;
	p = new point(1,2);
	p.x = 2 * p.y;
	...

Le système de types de C#, très proche du CTS, propose essentiellement d'utiliser des classes. La différence entre les deux premiers exemples est simplement le mécanisme d'accès aux coordonnées du point : par accès direct aux champs ou via des accesseurs (des propriétés). Il est également possible d'utiliser les structures C#, qui sont implantées au moyen de types valeurs. Les structures sont donc allouées sur la pile, passées par valeur et ne supportent pas de mécanisme d'héritage. Il est toutefois possible de déclarer des méthodes d'instance ou statiques au sein des structures.

Quelques implantations du point en Objective Caml

Déclaration	**Utilisation**

`class point (x0,y0) = object`
`   mutable val x = x0`
`   mutable val y = y0`
`   method get_x = x`
`   method set_x x' = x <- x'`
`   method get_y = y`
`   method set_y y' = y <- y'`
`end`
	`let p = new point (1,2) in`
	`   p#set_x (2 * p#get_y);...`

`type point = {`
`   mutable x:int;`
`   mutable y:int;`
`}`
	`let p = {x = 1; y = 2} in`
	`   p.x <- 2 * p.y;...`

Variant polymorphe : pas de déclaration	`` let p = `Point(1,2) in ``
	`let p' = match p with`
	`` `Point(x,y) -> `Point(2*y, y) in... ``

n-uplet : pas de déclaration	`let p = (1,2) in`
	`let p' = match p with`
	`   (x,y) -> (2 * y, y) in...`

En Objective Caml, le système de classes permet une implantation à la manière du deuxième exemple C#, avec manipulation des champs via des accesseurs. Il y a de nombreuses alternatives : les enregistrements, proches des structures C# (ils ne supportent pas l'héritage, cependant ils ont une sémantique de passage par pointeur), mais aussi des types anonymes, qui n'ont pas besoin de déclaration préalable. Les exemples précédents suggèrent une utilisation par partage et modification physique des valeurs, mais la copie est également possible et naturelle pour les objets et enregistrements Objective Caml.

De leur côté les variants polymorphes et les n-uplets permettent de manipuler des données hétérogènes précisément typées (contrairement à ce qu'un type anonyme comme `object[]` pourrait faire en C#), sans nécessiter de déclaration. Les possibilités d'interaction assurées par le type sont définies implicitement.

Le problème des expressions de types sans définition. En définitive, les types implantent à la fois restrictions et garanties. Les types nommés posent d'emblée une restriction (en établissant le domaine de compatibilité) et ensuite offrent des garanties (une valeur d'un type donné sera capable de prendre en charge telle ou telle opération).

Le fait que les types CTS reposent principalement sur des types déclarés crée une difficulté d'implantation : comment représenter les types purement algébriques de Caml ? Partant du constat que le système de types CTS est beaucoup moins souple quant à l'utilisation de types définis implicitement, deux solutions s'offrent à nous :

1. Implanter les types algébriques sans définition de Caml comme des instances d'un type CTS qu'il aura fallu définir au préalable.
2. Se passer de définition de type et se ramener aux quelques types définissables implicitement dans le CTS.

• Si l'on reprend l'exemple du point défini comme un n-uplet, la première solution consiste à définir dans le CTS une classe `int_times_int` définissant deux champs entiers et à utiliser des instances de cette classe partout où le type `int * int` de Caml est utilisé. Cela pose deux problèmes : la localisation de cette définition et l'intégration au polymorphisme paramétrique.

Premièrement, il faut prévoir de faire référence à ce type prédéfini partout dans le programme où le type `int * int` est apparent. Or dans le cadre d'une compilation modulaire, il est impossible a priori d'évaluer la portée d'un type et de plus on ne peut pas multiplier les définitions car elles seraient incompatibles entre elles. On peut alors imaginer de supporter ces définitions dans une bibliothèque d'exécution globale mais cela pose un problème d'exhaustivité : il n'est pas possible de fournir à l'avance des définitions CTS pour toutes les expressions de types possibles ! On a ici un problème de compilation séparée et d'édition de liens.

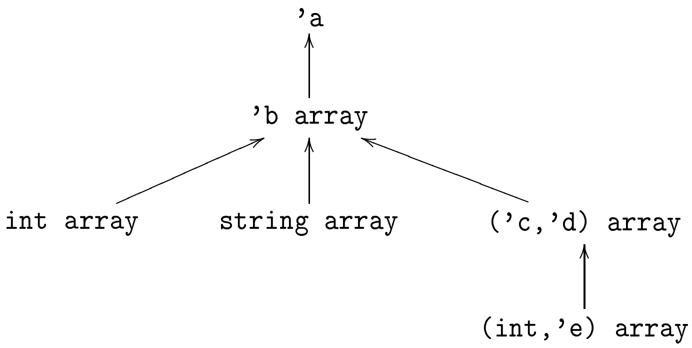
Deuxièmement, le gain obtenu par la définition d'une classe reproduisant fidèlement l'expression de type risque d'être perdue au profit d'une définition plus générique en raison des problèmes de polymorphisme paramétrique (il faut un moyen de mettre en relation le type `int_times_int` avec un type `object_times_object` par exemple, voir à ce sujet la section 2.1.2.2).

• L'autre solution consiste à utiliser les quelques types anonymes du CTS et y faire rentrer les valeurs Caml. Ainsi pour le point du plan on pourra utiliser `int[]` ou `object[]`. Remarquons que la deuxième de ces deux représentations est incontournable dès que le type contient des données hétérogènes, ce qui est un inconvénient majeur : l'accès aux valeurs internes du type structuré nécessitera des opérations de (dés)encapsulation. De plus les problèmes posés par le polymorphisme paramétrique subsistent (mettre en relation `int[]` et `object[]`).

En raison des difficultés posées par la première approche, nous penchons pour la deuxième solution, en dépit de ses inconvénients. L'implantation sera donc moins efficace pour les types n-uplets et les variants polymorphes que pour les enregistrements et les variants simples. Si l'utilisation des variants polymorphes reste assez marginale, c'est malheureusement moins le cas des n-uplets...

2.1.2.2 Au sujet du polymorphisme paramétrique

Le problème. Le polymorphisme paramétrique est une caractéristique essentielle des langages fonctionnels à la ML. Il est possible de définir des types paramétriques, comme `'a array`, le type des tableaux d'éléments de type `'a`. Cela conduit à considérer une relation d'instanciation entre types, par exemple :



Dans ce cadre, les termes et donc les valeurs ont un type *principal* : tout type acceptable pour ce terme placé dans un contexte licite est une instanciation de son type principal (pour les notions de type principal, d'unification et d'instance de types dans un formalisme décrivant le système de types de langages à la ML, on pourra consulter [64]). Par exemple :

Valeur	**Type principal**
`[|1;2|]`	`int array`
`[||]`	`'a array`
`let first tab = tab.(0)`	`'a array -> 'a`
`let sumints = Array.fold_left (+) 0`	`int array -> int`

Dans un contexte attendant une valeur de type `float array`, `[||]` est licite (`float array` est une instanciation de `'a array`) mais pas `[|1;2|]` (`float array` n'est pas une instanciation de `int array`).

D'autre part si l'on considère les exemples ci-dessus, on peut appliquer `first` et `sumints` indifféremment à `[|1;2|]` et `[||]`. Les deux fonctions sont respectivement confrontées aux deux arguments par le biais du contexte de l'application, qui lui-même a pour type principal `('a -> 'b) -> 'a -> 'b`, soit après unification :

Application	**Contexte unifié**	**Instanciation**
`first [|1;2|]`	`(int array -> int) -> int array -> int`	de `first`
`first [||]`	`('a array -> 'a) -> 'a array -> 'a`	*aucune*
`sumints [|1;2|]`	`(int array -> int) -> int array -> int`	*aucune*
`sumints [||]`	`(int array -> int) -> int array -> int`	de `[||]`

On voit que le polymorphisme implique l'instanciation des valeurs et que celle-ci dépend du contexte. Comment représenter les types paramétrés dans le CTS et gérer l'instanciation de valeurs Caml (tout en travaillant sans les Generics) ?

La solution. On peut d'ores et déjà faire les remarques suivantes :

- Si le CTS ne dispose que d'un polymorphisme d'interfaces, on peut tout de même s'appuyer sur l'héritage, qui permet *upcast* et *downcast* de valeurs. Le type `object` est à la racine de la hiérarchie des types références, et les types valeurs (si on est prêt à payer le coût des (dés)encapsulations) peuvent s'intégrer à cette hiérarchie.
- Les valeurs Caml peuvent très bien avoir plusieurs représentations dans le système de types CTS. La correspondance entre types Caml et types CTS n'est ni injective ni fonctionnelle. Deux types Caml peuvent être implantés au moyen d'un même type CTS et il également possible qu'une valeur Caml d'un type donné ait plusieurs représentations dans le CTS.

Les principales contraintes sont :

- Le respect de la sémantique : les valeurs structurées étant passées par référence, toutes les opérations modifiant le type CTS et qui procèdent par copie sont à proscrire. Par exemple le passage d'une valeur de type `int[]` à une valeur de type `object[]` ne peut se faire sans copie.
- La difficulté de spécialiser les valeurs dans le cadre de la compilation séparée : une approche de *monomorphisation* consiste à spécialiser le code d'une fonction polymorphe pour qu'elle s'adapte à ses différents contextes d'application. Le problème est de connaître à l'avance l'ensemble des monomorphisations utiles, ce qui est impossible à réaliser statiquement dans le cadre d'une véritable compilation séparée. L'information utile n'est connue intégralement qu'au moment de l'édition de liens [6].
- Une mise œuvre efficace : implanter une fonction générique capable de s'adapter à des types totalement différents est possible mais coûteux. Imaginons que l'implantation d'une fonction de type Caml `'a array -> 'a` soit capable de recevoir des valeurs de types `int[]` et `object[]` : déjà le type de l'argument de l'implantation sera nécessairement `object`, mais en plus une inspection dynamique du type de l'argument sera nécessaire pour exécuter le code adéquat.

Ces problèmes ne se posent pas dans l'environnement Caml car les valeurs ont une représentation uniforme (les entiers natifs) : le code des fonctions polymorphes reste générique dans un cadre non typé.

Alors que SML.NET a fait le choix d'une monomorphisation globale au moment de l'édition de liens, l'implantation de OCamIL repose sur une représentation générique des valeurs : nous évacuons les paramètres de types en utilisant le type CTS `object` partout dans le type Caml où la paramétricité s'exprime. D'autre part les représentations utilisant des types valeurs seront sujettes à des opérations de (dés)encapsulation. Ces choix découlent de la volonté de maintenir la compilation

6. Et encore dans le cas d'une bibliothèque de fonctions, l'édition de liens ne permet pas de résoudre la question. Une bibliothèque exportant une fonction polymorphe devra obligatoirement proposer une variante générique de cette fonction !

séparée comme dans le compilateur Caml standard (et de proscrire lors de l'édition de liens des opérations s'apparentant à de la compilation).

2.2 Annotation par les types

La génération de code CIL doit être guidée par une information de type qui a été perdue au cours des premières phases de compilation. Cette section se penche sur les deux méthodes (reconstruction et propagation) que nous avons testées dans le compilateur OCamIL afin de résoudre ce problème.

Ces deux méthodes, différentes par leur principe, varient par la quantité de modifications à apporter au compilateur Caml pour les mettre en œuvre. Elles s'inscrivent pourtant toutes les deux dans une architecture commune.

2.2.1 Un cadre commun

Les deux méthodes ont pu être implantées de manière modulaire au sein du prototype du compilateur OCamIL (un argument en ligne de commande permettant de passer de l'une à l'autre).

2.2.1.1 Une chaîne de compilation modulaire

On peut clairement distinguer trois grandes phases successives au cours des transformations menées par le compilateur OCamIL :

- **La phase initiale** (le *front-end*). Celle-ci comprend l'analyse lexicale et syntaxique ainsi que le typage. Elle est prise en charge par le code originel du compilateur Caml.
- **L'obtention d'un code `Clambda` décoré par les types**. Cette phase est soit obtenue par reconstruction soit par propagation de types.
- **La production de code CIL** (le *back-end*). La compilation de code CIL peut avoir lieu indépendemment de la méthode suivie lors des étapes précédentes. La première représentation intermédiaire lors de cette phase est le langage `ILM` (voir la sous-section 3.2.1.1).

L'objectif de la section 2.2 est de décrire la phase intermédiaire, figurée sur le diagramme 2.2. Rappelons que nous restons proches de l'implantation Caml d'origine pour suivre les évolutions du compilateur.

La réutilisation du code d'origine de Caml est maximale pour la méthode de reconstruction de types alors que la propagation nécessite d'intervenir dans tous les modules Caml permettant le passage de l'arbre de syntaxe typé `Typedtree` jusqu'au langage `Clambda`, afin de faire suivre les informations de types.

Le fonctionnement du compilateur repose sur l'interface entre la phase intermédiaire et le back-end, assurée dans tous les cas par un même langage intermédiaire `Ctypedlambda`. Le détail de la prise en charge de ce langage et de l'émission de code fait l'objet du chapitre 3.

$$
\begin{tikzcd}
 & & \texttt{Lambda} \arrow[r, "(C)"] & \texttt{Clambda} \arrow[d, "(R)"] & \\
\cdots \arrow[r, "(C)"] & \texttt{Typedtree} \arrow[ur, "(C)"] \arrow[dr, "(P)"'] & & \texttt{typeinfo Ctypedlambda} \arrow[r, "(B)"] & \texttt{ILM}\ldots \\
 & & \texttt{Typedlambda} \arrow[ur, "(P)"'] & &
\end{tikzcd}
$$

Légende :
- (C) compilateur Caml classique
- (R) branchement pour la reconstruction de types
- (P) branchement pour la propagation de types
- (B) back-end commun

FIG. 2.2 – *Les deux méthodes d'annotation de types*

2.2.1.2 Le langage intermédiaire annoté `Ctypedlambda`

L'objectif de la phase intermédiaire d'annotation de types est la production d'une représentation (`typeinfo Ctypedlambda`) :

- `Ctypedlambda` est une nouvelle représentation intermédiaire qui permet simplement de décorer une expression `Clambda` par des types. C'est un type Caml paramétré en la grammaire utilisée pour la décoration de types, ce qui permet de la réutiliser dans différents contextes.
- `typeinfo` est la grammaire de types retenue pour la décoration du code `Clambda` et que nous détaillons dans la suite.

Le langage `Ctypedlambda` et la grammaire de types `typeinfo`. Le langage `a' Ctypedlambda` est une simple extension de `Clambda` (déjà détaillée à la section 2.1.1.4) dont l'arbre de syntaxe est décoré à chaque nœud par une valeur de type `'a`. Ces emplacements sont utilisés pour ajouter des informations de types (on peut retrouver une expression `Lambda` par projection).

Le compilateur OCamIL utilise essentiellement une représentation des types appelée `typeinfo`.

Voici une présentation formelle d'un sous-ensemble de `typeinfo Ctypedlambda` que nous utiliserons à plusieurs reprises dans la suite. Les expressions typées sont notées $\mathbf{u}\colon \mathsf{t}$, où $\mathbf{u}$ est défini de la manière suivante :

$$
\begin{aligned}
\mathbf{u} := {} & \mathbf{x} \ \textit{(variables)} \\
& \mid \mathbf{let\ x = u_1}\colon \mathsf{t}_1 \ \mathbf{in\ u_2}\colon \mathsf{t}_2 \\
& \mid \mathbf{n} \ \textit{(constantes entières)} \\
& \mid \mathbf{closure} \begin{pmatrix} \mathbf{L_1},\mathbf{a_1},\mathbf{x_1^1}\colon \mathsf{t}_1^1 \ldots \mathbf{x_1^{k_1}}\colon \mathsf{t}_1^{\mathsf{k}_1},\mathbf{u_1}\colon \mathsf{t}_1 \\ \vdots \\ \mathbf{L_n},\mathbf{a_n},\mathbf{x_n^1}\colon \mathsf{t}_n^1 \ldots \mathbf{x_n^{k_n}}\colon \mathsf{t}_n^{\mathsf{k}_n},\mathbf{u_n}\colon \mathsf{t}_n \end{pmatrix} (\mathbf{u'_1}\colon \mathsf{t}'_1, \ldots, \mathbf{u'_m}\colon \mathsf{t}'_m) \\
& \mid \mathbf{offset_n}(\mathbf{x}\colon \mathsf{t})
\end{aligned}
$$

$$
\begin{array}{l}
\mid \mathbf{Dapply}(L,\mathbf{u_1}\colon \mathsf{t}_1,\ldots,\mathbf{u_n}\colon \mathsf{t_n}) \\
\mid \mathbf{Gapply}(\mathbf{u}\colon \mathsf{t},\mathbf{u_1}\colon \mathsf{t}_1,\ldots,\mathbf{u_n}\colon \mathsf{t_n}) \\
\mid \textbf{if } \mathbf{u}\colon \mathsf{t} \textbf{ then } \mathbf{u_1}\colon \mathsf{t}_1 \textbf{ else } \mathbf{u_2}\colon \mathsf{t}_2 \\
\mid \textbf{switch } \mathbf{u}\colon \mathsf{t} \textbf{ with } \begin{cases} \mathbf{i_1} \to \mathbf{u_1}\colon \mathsf{t}_1 \\ \vdots \\ \mathbf{i_n} \to \mathbf{u_n}\colon \mathsf{t_n} \\ \_ \to \mathbf{u_d}\colon \mathsf{t_d} \end{cases} \\
\mid \mathbf{catch_i}\ \mathbf{u_1}\colon \mathsf{t}_1 \textbf{ with } \mathbf{x_1}\colon \mathsf{t}_1^{\mathsf{x}} \ldots \mathbf{x_n}\colon \mathsf{t}_{\mathsf{n}}^{\mathsf{x}} \to \mathbf{u_2}\colon \mathsf{t}_2 \\
\mid \mathbf{fail_i}(\mathbf{u_1}\colon \mathsf{t}_1,\ldots,\mathbf{u_n}\colon \mathsf{t_n}) \\
\mid \mathbf{prim}(\mathbf{\Pi},\mathbf{u_1}\colon \mathsf{t}_1,\ldots,\mathbf{u_n}\colon \mathsf{t_n})
\end{array}
$$

Abstraction faite de l'ajout des informations de types, le langage est conforme à `Clambda` présenté à la section 2.1.1.4 (on pourra s'y reporter pour l'explication de chaque expression). Le sous-ensemble retenu prend en compte le noyau fonctionnel du langage (y compris l'application partielle et les fonctions mutuellement récursives), le filtrage de motifs et les valeurs structurées comme les variants et les enregistrements. Pour simplifier, les seules constantes sont les constantes entières.

Les types t définissent un sous-ensemble de `typeinfo` (dont la version complète sera donnée dans la section suivante) :

$$
\begin{array}{lll}
\mathsf{t} := & \mid \mathsf{int} \mid \mathsf{bool} & \textit{(types de base)} \\
& \mid \mathsf{block} \mid \mathsf{record}_{\mathsf{pa,dr}} \mid \mathsf{variant}_{\mathsf{pa,dv}} & \textit{(types structurés)} \\
& \mid \mathsf{t}_1 \to \ldots \to \mathsf{t_n} \to \mathsf{t} \mid \mathsf{clos} \mid \mathsf{sclos} & \textit{(types fonctionnels)} \\
& \mid \mathsf{obj} & \textit{(type généraliste)}
\end{array}
$$

Dans cette notation :

- pa désigne un chemin de type (un nom de type pleinement qualifié).
- La description $\mathsf{dr} = (\mathsf{x}_1 : \mathsf{t}_1, \ldots, \mathsf{x_n} : \mathsf{t_n})$ d'un enregistrement est la donnée de la suite de ses champs typés.
- La description $\mathsf{dv} = (\mathsf{C}_1 : (\mathsf{t}_1^1, \ldots, \mathsf{t}_1^{\mathsf{n}_1}) | \ldots | \mathsf{C_k} : (\mathsf{t}_{\mathsf{k}}^1, \ldots, \mathsf{t}_{\mathsf{k}}^{\mathsf{n_k}}))$ d'un variant est la donnée de la suite de ses constructeurs et pour chacun d'eux leur composition typée.

Cette grammaire limitée rend compte des entiers, des booléens, des blocs (sans plus de détail dans le cas du type block, ou plus spécifiquement pour les enregistrements et les variants) et des types fonctionnels (clos et sclos permettent de typer les fermetures et les fermetures partagées). Le type obj type les valeurs de manière générique.

Les primitives Π sont : $+$, $-$, $*$, $/$ (opérations arithmétiques), $==$, $!=$ (comparaisons), $Field_n$ (accès au champ d'un bloc) et $Block_{tag}$ (construction d'un bloc). Celles-ci sont toutes issues du langage `Clambda` original. Toutefois l'implantation de OCamIL ajoute la possibilité de décorer $Field_{n,tag}$ par un entier de *tag* supplémentaire, ce qui sera utile pour typer les accès aux champs des types variants (voir la section 2.2.3.2).

On se permettra d'alléger la notation **u**: `t` en **u** quand le contexte le permet.

La grammaire `typeinfo` dans le compilateur réel. La grammaire `typeinfo` complète utilisée par le compilateur OCamIL est définie en Caml comme suit :

```
type namedtype = {nt_ext:bool;nt_path:string list}

type typeinfo =
(* types de base *)
  | TIint | TIchar | TIbool | TIfloat | TIstring  | TIunit  | TIvoid
  | TIint32 | TIint64 | TInint
(* types structurés *)
  | TIblock  | TIrecord of namedtype | TIvariant of namedtype
  | TIarray of typeinfo | TItuple of typeinfo list | TIexception
  | TIlazy of typeinfo | TIlist of typeinfo | TIoption of typeinfo
(* types fonctionnels *)
  | TIarrow of typeinfo list * typeinfo | TIgenclosure | TIsharedclosure
(* divers *)
  | TIobject
  | TIpureIL of Il.typeref | TIdontknow
```

Types de base : tous les types de base de Caml se devaient de figurer dans la grammaire de types : c'est le cas avec `TIint`, `TIchar`, `TIbool`, `TIunit`, `TIstring`, `TIfloat`, `TIint32`, `TIint64` et `TInint`.

La grammaire distingue les types `TIunit` et `TIvoid` : cela permet de différencier les deux emplois de `unit` comme valeur ou comme absence de valeur. Le principe général est d'affecter le type `TIvoid` aux expressions qui ne laisseront pas de valeur empilée (selon le dispositif décrit à la section 1.3.2.1) lors de leur évaluation, alors que `TIunit` indique une valeur, même conventionnelle comme `null` (voir la section 3.1.1.1 pour plus de détails).

Remarquons que ceux qui sont considérés ici comme des types de base peuvent être aussi bien implantés par Caml au moyen d'entiers que de blocs.

Types structurés : la grammaire cherche à représenter les types structurés au moyen de différents degrés de précision : si `TIblock` désigne tout type de bloc sans distinction, les types `TIexception`, `TItuple`, `TIarray`, `TIrecord`, `TIvariant` rentrent dans le détail. Les types `TIrecord` et `TIvariant`, sur le modèle des expressions de types pour les types enregistrement et variant, font référence à un type nommé. Le booléen `nt_ext` est vrai lorsque le type est défini dans un module extérieur à celui dans lequel apparaît la référence. La suite de chaînes de caractères `nt_path` donne l'accès exact à la définition de types, en explicitant la suite des modules préfixant la déclaration du type[7].

La différence avec la formalisation ci-dessus est que la description des enregistrements et des variants n'est pas contenue dans l'expression de type elle-même. Une table maintenue par le compilateur associe les noms de types à leur description.

7. À titre de comparaison, dans le cas des expressions de types Caml, l'accès est implicite dans le cas général (dépendant des différents modules « ouverts » par la primitive `open` au niveau de la référence de type), nous y reviendrons à la section 2.2.3.3.

Notons enfin qu'on se donne la possibilité d'isoler certains types spécifiquement identifiés par le noyau du langage, comme `TIlazy`, `TIoption`, `TIlist` (ces derniers étant des types variants à part entière).

Types fonctionnels : les types fonctionnels trouvent écho dans le type `TIarrow`, complété par les types `TIgenclosure` et `TIsharedclosure` : les expressions manipulant des blocs fermetures (y compris partagées) sont introduites entre les langages intermédiaires `Lambda` et `Clambda` et n'ont pas nécessairement de type naturel associé au niveau du langage source.

Divers : le type `TIobject` sert à dénoter un type quelconque que la décoration de types n'a pas pu définir plus précisément. Ce type permet l'implantation du polymorphisme paramétrique conformément à ce qui a été argumenté dans la section 2.1.2.2. La fonction du type `TIdontknow` est différente puisqu'il est relatif à un type non encore identifié. Ce type est utilisé en interne lors des processus de décoration de types mais n'apparaît pas à l'interface avec la dernière phase de compilation.

Enfin, le type `TIpureIL` ouvre le système aux types de classes quelconques du CTS. Au sein du compilateur OCamIL, ces types sont vus comme des boîtes noires, car si des valeurs de ces types peuvent être passées, elles ne peuvent être inspectées par du code provenant de la compilation d'expressions Caml.

Discussion. Cette grammaire de types a été mise au point afin de refléter les types Caml mais elle est amenée à en différer pour les raisons suivantes :

- Toutes les expressions de `Clambda` n'ont pas nécessairement un type Caml naturel : le code source est transformé au cours des étapes de la compilation et les types qui ont un sens au niveau du programme source ne sont pas forcément adaptés au typage d'un langage intermédiaire.
- Afin de permettre l'interopération, il doit être possible de gérer des types CIL « purs », n'ayant pas de rapport avec les types choisis pour représenter les valeurs Caml. Même si les programmes Caml ne sont pas amenés à intervenir spécifiquement sur des valeurs transmises par un composant externe, écrit dans un autre langage, celles-ci doivent pouvoir être manipulées à travers des constructions génériques telles que des fonctions polymorphes ou des types abstraits. En conséquence, la grammaire de types ne doit pas traiter exclusivement les représentations Caml.
- Une fois au stade des représentations, il n'est pas toujours utile de conserver une information de typage aussi complète que celle exprimée par le système de type de Caml. Toute sa richesse n'est pas forcément exploitable et d'autre part une partie de l'information de type n'est parfois uniquement utile que pour les besoins de l'inférence et de la vérification de types.
- La grammaire doit pouvoir se plier au traitement du polymorphisme paramétrique tel qu'il a été exposé lors de la discussion de la section 2.1.2.2.
- La grammaire doit permettre une approximation des types Caml dans l'éventualité d'une perte partielle d'informations. Nous verrons qu'il n'est pas tou-

jours possible d'inférer la nature exacte des types mis en jeu derrière les expressions `Clambda`.

2.2.2 Synthèse de la représentation annotée

L'idée de reconstruire les types est conforme aux principes énoncés pour OCamIL à la section 1.3.2.2, car cette approche permet de retarder le plus possible l'insertion de nouveau code sur le compilateur Caml. On s'attend bien sûr en retour à devoir faire des concessions sur le degré de précision de l'information reconstituée, ce qui n'est pas sans conséquence sur les performances.

Nous pouvons envisager plusieurs algorithmes de reconstruction de types dont l'objectif est d'insérer partout où c'est possible des informations de typage `typeinfo`. Ils travaillent typiquement sur une instanciation de `'a Ctypedlambda`. Les algorithmes peuvent être plus ou moins sophistiqués, mais tous se basent sur l'analyse des constantes et des primitives utilisées dans le code `Clambda`.

La synthèse d'information de types est de toutes façons tributaire des choix d'implantation de Caml en matière de primitives, et en particulier ne peut extraire davantage d'information que contenue dans celles-ci. La différence entre les algorithmes de reconstruction de types réside dans les moyens mis en œuvre pour propager les informations partout dans le code, et en particulier la richesse de la grammaire de types utilisée pour incarner ces informations.

2.2.2.1 Exploitation des primitives

Nous présentons dans la suite quelques exemples d'expressions Caml et du code `Clambda` associé qui illustrent les moyens d'extraire les types des primitives et des constantes. Par souci d'homogénéité, on emploiera les conventions typographiques utilisées pour la formalisation de `Ctypedlambda` y compris pour des constructions qui ne font pas partie du noyau formalisé.

Exemple 1. Considérons l'expression suivante :

```
let reverse s10 =
  for i = 0 to 4 do
    let c = s10.[9-i] in
      s10.[9-i] <- s10.[i];
      s10.[i] <- c;
  done
```

L'arbre `Clambda` engendré pour cette expression s'écrit :

let reverse =
  closure(Lreverse,1,s10,
    for i = **0 to 4 do**
      let c = **prim(string.get,s10,prim(**$-$**,9,i)) in**
        (**seq**
          prim(string.set,s10,prim($-$**,9,i),prim(string.get,s10,i)))**
          prim(string.set,s10,i,c)

```
      )
    done
  ) in ...
```

Les constantes et primitives présentes dans cet exemple fournissent les informations suivantes :

- **0**, **4** et **9** sont des constantes de type entier.
- La primitive $-$ a pour type : $\mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$.
- **string.get** est une primitive de type : $\mathsf{string} \rightarrow \mathsf{int} \rightarrow \mathsf{char}$.
- **string.set** est une primitive de type : $\mathsf{string} \rightarrow \mathsf{int} \rightarrow \mathsf{char} \rightarrow \mathsf{void}$.

L'examen de l'expression permet de rassembler progressivement les informations suivantes :

1. **reverse** est un bloc fermeture. La fermeture n'a pas d'environnement et contient une fonction d'étiquette **Lreverse** ayant un argument. La fonction a donc pour type $\mathsf{t}_1 \rightarrow \mathsf{t}_2$, où t_1 est le type de l'argument **s10**.
2. **i** est de type entier.
3. **c** est de type char et **s10** est de type string. Ainsi $\mathsf{t}_1 = \mathsf{string}$.
4. L'intérieur de la boucle **for** est cohérent du point de vue des types inférés. La boucle elle-même ne retourne pas de valeur (elle a pour type void). Ainsi $\mathsf{t}_2 = \mathsf{void}$.

L'expression `Ctypedlambda` ainsi calculée s'écrit :

```
let reverse: clos =
  closure(Lreverse,1,s10: string,
    for i: int = 0: int to 4: int do
      let c: char = prim(string.get,s10: string,prim(−,9: int,i: int): int) in
        (seq
          prim(string.set,s10: string,
                              prim(−,9: int,i: int): int,
                              prim(string.get,s10: string,i: int): char): void
          prim(string.set,s10,i,c): void
        ): void
    done: void
  ) in ...
```

La variable **reverse** a reçu le type clos. Dans ce cas particulier, puisque la fermeture est réduite à une fonction on pourrait lui attribuer le type fonctionnel $\mathsf{string} \rightarrow \mathsf{void}$. Mais ce n'est de toutes façons pas nécessaire car les informations de type sur chaque élément de la fermeture (fonction ou élément de l'environnement) sont conservées dans l'arbre `Ctypedlambda` au niveau de la sous-expression **closure**.

L'utilisation de void vient du fait que les expressions concernées ne laisseront pas de valeur sur la pile lors de leur évaluation (selon le dispositif décrit à la section 1.3.2.1).

Exemple 2.

```
let filtrage = function
  | (0,(x,0)) | (_,(0,x)) -> x
  | (x,(y,_)) when x = y -> x
  | (x,(y,z)) when y != z -> x + y + z
  | (_,(y,_)) -> y
```

La compilation du filtrage de motifs par le front-end Caml modifie profondément l'allure des expressions :

```
01  let filtrage =
02      closure(Lfiltrage,1,
03        param,
04          let x = prim(Field_0,param) in
05            catch_2
06              if prim(! =_i ,x,0) then fail_2 else
07                let match = prim(Field_1,param) in
08                  if prim(! =_i ,prim(Field_1,match),0) then
09                    fail_2 else prim(Field_0,match)
10            with
11              let match = prim(Field_1,param) in
12              let y = prim(Field_0,match) in
13                if prim(! =_i ,y,0) then
14                  if prim(==_i ,x,y) then x else
15                    let z = prim(Field_1,match) in
16                      if prim(! =_i ,y,z) then
17                        prim(+,prim(+,x,y),z) else y
18                else prim(Field_1,match)
19  ) in ...
```

Dans le code précédent, **param** désigne le motif initial à filtrer et **match** le sous-motif de droite. La construction **catch** ... **with** cherche à isoler le cas (0,(x,0)). En cas d'échec les différents cas restants sont analysés dans des structures **if** ... **else**.

L'information de types est extraite à partir des constantes entières et des primitives : +: int → int → int, ainsi que $==_\mathbf{i}$ et $! =_\mathbf{i}$ de type int → int → bool (ce ne sont pas les comparaisons polymorphes, mais les comparaisons spécialisées aux entiers : le compilateur Caml se sert toujours des types pour optimiser les comparaisons), et la primitive **field**: block → int → object. Cette dernière est utilisée indifféremment pour de nombreux types de blocs, et elle retourne un entier machine : on n'a pas la possibilité à première vue de savoir si elle rend un entier ou bloc par exemple.

En ce qui concerne la synthèse des types :

1. À la ligne 4, on ne peut directement déterminer le type de **x**, bien qu'on apprenne que **param** est un bloc. L'information arrive à l'intérieur du **catch**, à la ligne 6 grâce à la comparaison de **x** et d'un entier.

2. Le type de l'expression formée des deux **if** imbriqués entre les lignes 6 et 9 n'apparaît pas immédiatement, car les deux **fail** n'ont pas de type intrinsèque et **prim**(**field**$_0$,**match**) ne nous aide guère.
3. Tout s'éclaire entre les lignes 11 et 18 : **y** est un entier (cf. ligne 13) donc **z** aussi (ligne 16), ce qui est confirmé par les additions de la ligne 17. L'expression qui s'étend entre les lignes 11 et 18 est donc entière, information qui remonte de certaines branches des **if** (ligne 14 et 17, mais pas 18 directement).
4. On peut identifier le type (inconnu) de l'expression à l'intérieur du bloc **catch** à celui du bloc **with** : un entier.

On peut finalement décorer par les types de la manière suivante (par souci de clarté on ne donne pas obligatoirement les types de chaque sous-expression) :

```
let filtrage: clos =
    closure(Lfiltrage,1,
      param: block,
        let x: int = prim(Field0,param: block) in
          catch2
            if prim(! =i ,x: int,0): bool then fail2: int else
              let match: block = prim(Field1,param: block) in
                if prim(! =i ,prim(Field1,match: block),0): bool then
                  fail2: int else prim(Field0,match: block): int
          with
            let match: block = prim(Field1,param: block) in
            let y: int = prim(Field0,match: block) in
              if prim(! =i ,y: int,0): bool then
                if prim(==i ,x: int,y: int): bool then x: int else
                  let z: int = prim(Field1,match: block) in
                    if prim(! =i ,y: int,z: int): bool then
                      prim(+,prim(+,x: int,y: int),z: int) else y: int
              else prim(Field1,match: block): int
) in ...
```

Les variables **param** et **match** reçoivent le type block. Il n'est pas toujours possible de deviner le type exact qui se cache derrière les blocs, et même quand c'est possible, comme dans l'exemple précédent (où on pourrait utiliser une grammaire de types de blocs avec un polymorphisme de rangée, qui typerait **match**: [int,int,..] et **param**: [int,[int,int,..],..]) le CTS ne fournit pas la possibilité de représenter ces types fortement algébriques de manière anonyme, ou bien à la fois nommée et cohérente à travers la frontière des modules, comme indiqué dans la section 2.1.2.1.

Exemple 3. Regardons maintenant le cas d'un type variant paramétré et d'une fonction polymorphe associée.

```
type 'a tree =
  | Node of 'a tree * 'a tree
  | Leaf of 'a
```

```
let subst a b tree =
  let rec srec = function
    | Node(t1,t2) -> Node(srec t1,srec t2)
    | Leaf c as l -> if c = a then Leaf b else l
  in
    srec tree

let nospace = subst ' ' '_'
let nozero = subst 0.0 0.0001
```

La fonction `subst` est polymorphe de type `'a -> 'a -> 'a tree -> 'a tree` et les fonctions `nospace` et `nozero` en sont des applications partielles et monomorphes. L'arbre `Clambda` produit pour l'ensemble du code ci-dessous est :

$$
\begin{array}{l}
\textbf{let subst } = \\
\quad \textbf{closure}(\textbf{Lsubst},\textbf{3},(\textbf{a},\textbf{b},\textbf{tree}), \\
\qquad\qquad \textbf{let clos } = \\
\qquad\qquad\quad \textbf{closure}(\textbf{Lsrec},\textbf{1},(\textbf{l},\textbf{env}), \\
\qquad\qquad\qquad \textbf{switch l with} \\
\qquad\qquad\qquad \left\{ \begin{array}{ll}
\mathbf{0} \rightarrow & \begin{array}{l} \textbf{prim}(\textbf{Block}_\mathbf{0}, \\ \quad (\textbf{Dapply}(\textbf{Lsrec},\textbf{prim}(\textbf{Field}_\mathbf{0},\textbf{l}),\textbf{offset}_\mathbf{0}(\textbf{env}))) \\ \quad (\textbf{Dapply}(\textbf{Lsrec},\textbf{prim}(\textbf{Field}_\mathbf{1},\textbf{l}),\textbf{offset}_\mathbf{0}(\textbf{env}))) \end{array} \\
 & \\
\mathbf{1} \rightarrow & \begin{array}{l} \textbf{if prim}(==,\textbf{prim}(\textbf{Field}_\mathbf{0},\textbf{l}),\textbf{prim}(\textbf{Field}_\mathbf{2},\textbf{env})) \\ \quad \textbf{then prim}(\textbf{Block}_\mathbf{1},\textbf{prim}(\textbf{Field}_\mathbf{3},\textbf{env})) \textbf{ else l} \end{array}
\end{array} \right. \\
\qquad\qquad\quad)\ (\textbf{a},\textbf{b})\ \textbf{in} \\
\qquad\quad \textbf{let srec } = \textbf{offset}_\mathbf{0}(\textbf{clos})\ \textbf{in} \\
\qquad\qquad \textbf{Dapply}(\textbf{Lsrec},\textbf{tree},\textbf{srec})\ \textbf{in} \\
\textbf{let nospace } = \textbf{Gapply}(\textbf{subst},\text{' '},\text{'\_'})\ \textbf{in} \\
\textbf{let nozero } = \textbf{Gapply}(\textbf{subst},\textbf{0.0},\textbf{0.0001}) \\
\textbf{in} \ldots
\end{array}
$$

Bien que le type `tree` soit nommé et défini dans ce même programme source, il n'y a aucune trace de sa définition dans le code produit. Cet exemple a recours à des fermetures avec environnement. L'instruction **offset** prend une fermeture et renvoie une fermeture. On remarque ici que la primitive **Field** est également utilisée sur des blocs fermetures pour accéder à la partie environnement (voir l'action associée au cas de tag 1). La primitive $\mathbf{Block_i}$ a un type qui dépend du nombre de ses arguments mais qui est toujours de la forme $\mathsf{obj} \rightarrow \ldots \rightarrow \mathsf{obj} \rightarrow \mathsf{block}$. En ce qui concerne le retypage :

1. Les types de **subst**, de **clos** et de **env** sont déterminés immédiatement ; ce sont des fermetures. Le type de **l** est donné par le **switch** : il s'agit d'un type variant donc de obj (les variants peuvent être des entiers ou des blocs). Il est possible de raffiner de la manière suivante : l'analyse des cas **switch** montre que seuls des constructeurs non constants sont traités : comme le compilateur Caml génère toujours du code pour les cas par défaut, cela donne la certitude que **l** est toujours un bloc; ce qui permet de typer par block.

2. La confrontation des différents cas permet de déduire le type de retour de **Lsrec** : dans le cas 0, on a un bloc et dans le cas 1, un bloc ou un objet suivant la finesse de l'analyse précédente, ce qui au final donne un bloc ou un objet.
3. Il est possible de maintenir un environnement de typage qui fait le lien entre les champs environnement de la fermeture courante et les instructions **Field** utilisées pour y accéder, c'est-à-dire dans notre exemple d'identifier les types de **prim**(**Field$_\mathbf{2}$**,**env**) et de **prim**(**Field$_\mathbf{3}$**,**env**) à ceux de **a** et **b**. Dans le cas présent cela ne nous apprend rien de plus sur **a** et **b** qui reçoivent le type obj.
4. Ensuite **srec** est typé comme une fermeture et **tree** comme un objet (ou un bloc en cas d'analyse fine). La fonction associée à la fermeture **subst** peut alors recevoir dans le meilleur de cas le type obj → obj → block → block.
5. Enfin **nospace** et **nozero** reçoivent le type obj. En effet de manière à traiter tous les cas d'application (totale et partielle), l'application générique d'arité n n'a pas de type plus fin que **Gapply**: $\mathsf{clos} \rightarrow \underbrace{\mathsf{obj} \rightarrow \ldots \rightarrow \mathsf{obj}}_{n} \rightarrow \mathsf{obj}$.

Il n'est pas simple dans le cas général de typer plus finement les applications partielles et cela supposerait une analyse du code poussée.

D'autre part il est important de noter que le typage des fermetures doit être mené localement au code de cette fermeture (ou des fermetures en cas de fermetures partagées), même si l'information qu'on peut en tirer semble lacunaire. Ainsi l'information sur le prototype d'une fonction ne doit pas être affinée par les utilisations faites de la fonction, même si cela semble donner des compléments sur son type. On risquerait alors de monomorphiser une fonction polymorphe ce qui la rendra incompatible dans d'autres contextes. Dans notre exemple, il ne faut pas chercher à affiner le type de **subst** en char → char → block → block...

2.2.2.2 Panorama des difficultés rencontrées

Il est d'autant plus problématique de reconstruire les types Caml que :

- Ceux-ci forment un langage très expressif. L'information à reconstituer est loin d'être triviale et le simple examen du code `Clambda` ne garantit pas de pouvoir retrouver tous les types.
- Le compilateur Caml génère très tôt du code orienté implantation. Il est alors difficile de lire les types sur des primitives qui sont indifféremment utilisées pour accéder à des valeurs de types radicalement différents.

Polymorphisme des primitives et fusion des types par l'implantation. L'environnement d'exécution de Caml utilise la même structure de bloc pour différents types de valeurs, si bien que les mêmes primitives sont utilisées pour des types différents : cela interdit bien souvent de reconstituer les types d'origine derrière ces valeurs, au simple examen du code. L'algorithme de retypage ne pourra pas cibler la grammaire `typeinfo` dans sa totalité.

L'exemple type de primitive « polymorphe » est donné par **Field** et **SetField**, accédant en lecture et en écriture aux blocs Caml. Elles sont utilisées indifféremment sur tous les types de blocs, exceptés les flottants, les grands entiers et les chaînes de caractères. Par exemple les trois fonctions suivantes aboutissent toutes rigoureusement au même code `Lambda` (on suppose définis un type enregistrement `type r = {x:int; y:int}` et un type variant `type v = Pair of int * int`) :

```
let access1 = function {x=a;y=_} -> a
let access2 = function Pair(a,_) -> a
let access3 = function (a,_) -> a
```

Dans les trois cas on fait appel à **prim**(**Field$_0$**,**arg**) où **arg** est l'argument de la fonction. En revanche les deux fonctions suivantes ont recours à des primitives spécialisées (respectivement sur les chaînes de caractères et les tableaux) et permettent un typage plus précis :

```
let access4 = function s -> s.[0]
let access5 = function t -> t.(0)
```

De plus la primitive **Field** est également utilisée pour accéder au contenu de l'environnement d'une fermeture.

Les blocs ne sont pas les seuls à être source d'ambiguïté puisque les entiers Caml (au sens de l'environnement d'exécution Caml) implantent à la fois les entiers, les booléens et les caractères. On peut bien souvent les distinguer grâce au contexte, mais cela n'est pas exhaustif. Ainsi une constante entière peut en fait cacher un booléen ou un caractère ! Il faut être particulièrement attentif aux opérations d'encapsulation et de désencapsulation des valeurs de type entier (voir la section 3.1.1.1).

Enfin la distinction fondamentale entre entiers et blocs Caml est parfois malmenée, comme expliqué dans la section suivante, consacrée aux types variants.

Le problème des types variant. Les variants posent un problème particulier car ils sont implantés à la fois au moyen d'entiers et de blocs Caml, ce qui est non-uniforme du point de vue du CTS.

Prenons l'exemple d'une définition simple de type variant :

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

Le variant `t` déclare deux constructeurs constants et un constructeur non constant. Pour l'environnement d'exécution de Objective Caml, ceux-ci sont respectivement représentés par les entiers `0`, `1` et un pointeur sur un bloc contenant une autre valeur de type `t`. C'est homogène pour l'environnement Caml mais l'algorithme de retypage peut éventuellement inférer deux types différents, int et block, pour des valeurs de type `t`.

Si on considère la fonction suivante et le code associé dans la représentation `Clambda` :

code Objective Caml	code `Clambda`

`let cut = function`
`  | Node n -> n`
`  | x -> x`
	let cut = **closure**(**Lcut**,**1**,**x**,
	if prim(**IsInt**,**x**) **then x**
	else prim(**Field$_0$**,**x**))

type : `t -> t`	type inféré : obj → obj

La primitive **IsInt** teste le bit de tag qui distingue les entiers des blocs : son type est obj → bool. La fonction **cut** ci-dessus est naturellement typée par obj → obj. Le type obj recouvre bien les grandes classes de valeurs Caml : entiers et blocs. Pourtant quelque-chose ne va pas, comme en témoigne l'exemple suivant :

code Objective Caml	code `Clambda`

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

type : `t -> t -> t`	type inferé : obj → int → int

Le type du paramètre **b** est problématique. À la vue du code source Objective Caml nous savons que **a** et **b** sont tous les deux de type `t`, mais à l'examen du code `Clambda`, on est tenté d'affirmer que **b** est un entier ! La seule information que l'algorithme de synthèse de types possède sur **b** est que son type est unifiable avec int (de par la sous-expression : **if prim(! $=_{\mathrm{i}}$,a,0) then b else 1**). Si on veut assurer un minimum de précision (c'est-à-dire ne pas tout typer par obj), **b** doit recevoir le type entier, et la fonction **problem** prend le type obj → int → int. Cette reconstruction de types incorrecte est désastreuse, et à des degrés divers.

Tout d'abord compiler une application de la forme `problem One (Node Zero)` conduit à la détection d'une incompatibilité par le vérificateur de types, car la valeur `Node Zero` n'a pas un type compatible avec int : cela interrompt brutalement la compilation. Ceci ne peut pas être évité par une politique de backtrack visant à corriger le type de **b**, car la définition de **problem** et son application peuvent être dans des modules distincts, donc compilés séparément.

Dans le cas précédent, la compilation échoue avec une erreur non récupérable. Cela peut être encore pire, par exemple si on introduit la fonction `f` suivante :

code Objective Caml	code `Clambda`

`let rec f i =`
`  if i==1`
`    then Node Zero` else
`    problem Zero (f(i-1))`
	let f = closure(Lf,1,i,
	if prim($==_{\mathrm{i}}$,i,1)
	then prim(Block$_0$,0)
	else Dapply(Lproblem,
	0,Dapply(Lf,prim(−,i,1))))

type : `int -> t`	type inféré : int → obj

L'information de types qui est reconstruite pour **f** est apparemment correcte. En réalité le type de retour obj provient de l'union entre les types des deux branches de l'instruction **if**, la première de type block et la seconde de type int issue du type (erroné) de **problem**. Plus tard, lors de l'émission de code, une instruction de désencapsulation sera insérée autour de l'appel récursif à **f** afin de convertir son résultat de type obj en entier.

Cette fois, la compilation d'un terme `let _ = f 2` réussit (l'instruction de conversion de obj à int est licite en général) mais produit du code incorrect. L'erreur ne sera décelée qu'à l'exécution, avec la levée d'une exception (d'incompatibilité de types) par l'environnement .NET à l'évaluation de l'application `f 2` : en effet elle conduit à l'application `problem Zero (Node Zero)` pour laquelle la conversion de types est incorrecte.

<u>Remède.</u> Nous proposons une solution simple qui passe par une légère modification de la représentation des types variant. Partant de la remarque qu'il est possible de représenter les constructeurs constants comme des blocs vides (dont le tag code le constructeur), nous représentons les types variants uniformément par des blocs. Cela permet d'éviter les conflits entre types blocs et types entiers qui causent les erreurs de retypage illustrées plus haut.

Quelques remarques :

- Il est facile d'implanter cette modification qui ne nécessite que quelques lignes de code, mais cela nécessite d'intervenir en amont sur la chaîne de compilation, ce qui contrarie la conception sous forme de back-end annoncée dans la section 1.3.2.2.
- Ce changement élimine tout utilisation de la primitive **IsInt** et permet de raffiner automatiquement le typage de l'instruction **switch** qui ne travaille plus que sur des blocs.
- Modifier la représentation de valeurs peut être sensible sur certains programmes Caml qui font implicitement des suppositions sur ces représentations (même si ce ne peut pas être considéré comme un exemple à suivre, ce type de programme existe, y compris dans la distribution Caml).
- On peut imaginer une politique de représentation plus compliquée, où les types variants constitués uniquement de constructeurs constants gardent une implantation au moyen d'entiers et tous les autres par des blocs comme proposé ci-dessus. Cela ne convient aux variants polymorphes qui peuvent être localement uniquement constitués de constructeurs constants mais rassembler des constructeurs non constants dans un autre contexte.

Si nous revenons à l'exemple de la fonction **problem**, le code `Clambda` produit pour celle-ci est maintenant :

$$\textbf{let problem} \;=\; \textbf{closure}(\textbf{Lproblem},\textbf{1},\textbf{a},\textbf{b},\textbf{switch a with} \begin{cases} 0 \rightarrow prim(Block_1) \\ 1 \rightarrow b \\ 2 \rightarrow b \end{cases})$$

Ainsi le type de la fonction, $\mathsf{block} \rightarrow \mathsf{block} \rightarrow \mathsf{block}$, n'est plus un problème ! Nous avons décidé d'implanter cette solution dans le compilateur OCamIL.

Manipulations du système de type. La synthèse de types peut être entravée par certaines manipulations du système de types permises par le langage Caml. Ainsi :
• *Les annotations de types explicites*, qui permettent de guider le typeur, ne peuvent être exploitées par le moteur de retypage de OCamIL car ces annotations ne laissent aucune trace au niveau des représentations intermédiaires.

• Plus problématiques sont les *contournements du système de types* effectués à l'aide du module (non documenté) `Obj` de la bibliothèque standard, et en particulier par la fonction `Obj.magic:'a -> 'b` déjà évoquée à la section 2.1.1.3. Celle-ci est implantée par la primitive « identité » et a pour but de casser le système de types. Alors que l'identité conduit naturellement le retypeur à identifier les types de son argument et de son résultat, il ne faut surtout pas le faire dans le cas où elle implante `magic`. Nous avons contourné ce problème en modifiant l'implantation de `magic` par une primitive **BoxedIdentity**, pour laquelle le typeur utilise le type `obj` à l'entrée et la sortie. Cela ne conduit pas dans le cas général au retypage le plus adéquat, mais donne des résultats corrects puisque le type `obj` a pour fonction de recouvrir tous les types.

2.2.3 Alternative : propagation des types

Le retypage du code intermédiaire `Clambda` est suffisant pour générer du code correct mais n'est pas assez précis. Nous verrons aux cours des chapitres suivants qu'il n'est pas possible dans ces conditions d'utiliser des représentations efficaces pour les valeurs Caml et l'efficacité des programmes produits s'en ressent.

Nous avons expérimenté une autre voie au cours du projet OCamIL : propager les informations de typage le long des représentations intermédiaires depuis la sortie de l'algorithme de typage du compilateur Objective Caml jusqu'au code `Clambda`. Cela suppose un plus grand nombre d'interventions en amont de ce code, et maintenir OCamIL pour être à la pointe des derniers développements de Objective Caml sera plus difficile : les types et leur implantation sont susceptibles d'évoluer avec la sortie de nouvelles versions du compilateur mais comme expliqué dans la section précédente, une implantation strictement sous la forme d'un back-end atteint rapidement ses limites de toutes façons.

La propagation des types permet de choisir des représentations plus fines, ce qui promet de meilleures performances mais a également d'autres applications, comme le débogage de programmes Objective Caml : en effet le code CIL engendré sera plus proche du code source.

2.2.3.1 Architecture de la propagation

Étapes de la propagation. Au sein du compilateur Objective Caml classique les phases suivantes font passer de l'arbre de syntaxe typé à la représentation `Clambda` :

$$\texttt{Typedtree} \xrightarrow{\substack{\texttt{Transl*}\\ \texttt{Matching}\\ \texttt{Simplif}}} \texttt{Lambda} \xrightarrow{\texttt{Closure}} \texttt{Clambda}$$

Nous avons indiqué les principaux modules du compilateur Caml qui sont responsables de chacune des transformations.

– La première phase construit la représentation `Lambda`[8]. La génération du code

8. Rappelons que celle-ci est similaire à `Clambda` mis à part que les fermetures ne sont pas explicitées. Le noyau fonctionnel est représenté à l'aide d'un opérateur d'abstraction et d'un opérateur d'application à la manière du λ-calcul.

`Lambda` est prise en charge par les modules `Translcore` (transformation des expressions du langage), `Translmod` (compilation des expressions du langage des modules), ainsi que `Translobj` et `Translclass` qui s'occupent spécifiquement de la compilation des objets et des classes Objective Caml. Le module `Translcore` délègue la compilation du filtrage de motifs au module `Matching`, qui effectue en réalité le plus gros travail.

– La deuxième phase passe par le module `Simplif` visant à alléger la représentation `Lambda` (principalement par des simplifications sur les constructions `let`), suivi par le module `Closure` qui calcule la représentation `Clambda` par l'explicitation de la gestion des fermetures.

Il nous faut intervenir dans l'ensemble de ces modules afin de propager une version typée de chaque représentation intermédiaire :

$$\texttt{Typedtree} \xrightarrow{\substack{\texttt{Transl*}\\ \texttt{Matching}\\ \texttt{Simplif}}} \texttt{Typedlambda} \xrightarrow{\texttt{Closure}} \texttt{CTypedlambda}$$

Outre les trois représentations manipulées au cours de ces transformations, deux langages de types différents sont utilisés :

1. Le langage de types d'origine de Objective Caml (en tout cas sa représentation interne au compilateur) est disponible dans `Typedtree` et sera propagée dans `Typedlambda`.
2. Le langage `typeinfo` (avec la représentation `Ctypedlambda`, déjà présentée plus haut) assure l'interface avec les étapes ultérieures de la compilation dans OCamIL. Il est la cible d'une transformation prenant en entrée le langage de types précédent.

La grammaire de types de Caml est la plus complète mais présente certaines caractéristiques qui ne sont utiles que lors de l'inférence de types. La grammaire `typeinfo` est moins précise mais tournée vers la représentation des valeurs Caml dans la machine cible. En plus de son rôle d'interface elle a l'avantage de la concision.

Nous détaillons maintenant les représentations intermédiaires.

Arbre de syntaxe typé `Typedtree`. Ce langage s'articule autour d'expressions formant le noyau du langage Caml et la description des motifs de filtrage.

Nous utiliserons une formalisation d'un sous-ensemble de `Typedtree` introduite ci-après. Les expressions typées sont notées[9] $\mathtt{e} : \tau$, où `e` est défini de la manière suivante :

9. On se permettra d'alléger la notation $\mathtt{e} : \tau$ en `e` quand le contexte le permet.

$$
\begin{array}{ll}
\mathtt{e} := \mathtt{x} & \textit{(variable)} \\
\quad \mid \mathtt{let} \left\{ \begin{array}{l} p_1 = \mathtt{e_1} : \tau_1 \\ \vdots \\ p_n = \mathtt{e_n} : \tau_n \end{array} \right\} \mathtt{in}\ \mathtt{e} : \tau & \textit{(liaison, cf. le langage de motifs)} \\
\quad \mid \mathtt{n} & \textit{(constante entière)} \\
\quad \mid \{\mathtt{x_1} = \mathtt{e_1} : \tau_1, \ldots, \mathtt{x_n} = \mathtt{e_n} : \tau_n\} & \textit{(enregistrement)} \\
\quad \mid \mathtt{C}(\mathtt{e_1} : \tau_1, \ldots, \mathtt{e_n} : \tau_n) & \textit{(variant)} \\
\quad \mid \mathtt{fun} \left\{ \begin{array}{l} p_1 \rightarrow \mathtt{e_1} \\ \vdots \\ p_n \rightarrow \mathtt{e_n} \end{array} \right\} : \tau \rightarrow \tau' & \textit{(abstraction, cf. le langage de motifs)} \\
\quad \mid \mathtt{let}^* \left\{ \begin{array}{l} x_1 = \mathtt{fun}\ \{\ldots\} \\ \vdots \\ x_n = \mathtt{fun}\ \{\ldots\} \end{array} \right\} \mathtt{in}\ \mathtt{e} : \tau & \textit{(fonctions mutuellement récursives)} \\
\quad \mid \mathtt{apply}\ (\mathtt{e} : \tau, \mathtt{e_1} : \tau_1, \ldots, \mathtt{e_n} : \tau_n) & \textit{(application)} \\
\quad \mid \mathtt{if}\ \mathtt{e} : \tau\ \mathtt{then}\ \mathtt{e_1} : \tau_1\ \mathtt{else}\ \mathtt{e_2} : \tau_2 & \textit{(conditionnelle)} \\
\quad \mid \mathtt{match}\ \mathtt{e} : \tau\ \mathtt{with} \left\{ \begin{array}{l} p_1 \rightarrow \mathtt{e_1} \\ \vdots \\ p_n \rightarrow \mathtt{e_n} \end{array} \right\} : \tau' & \textit{(filtrage, cf. le langage de motifs)} \\
\quad \mid \mathtt{prim}_\Pi & \textit{(primitive)}
\end{array}
$$

Les valeurs mutuellement récursives sont restreintes dans cette présentation aux valeurs fonctionnelles.

Le langage de motifs (lui-même muni de certaines informations de types) suit la définition suivante :

$$
\begin{array}{ll}
p := \_ & \textit{(motif universel)} \\
\quad \mid x & \textit{(variable)} \\
\quad \mid n & \textit{(entier)} \\
\quad \mid C(p_1 : \tau_1^p, \ldots, p_n : \tau_n^p) & \textit{(variant)} \\
\quad \mid \{x_1 = p_1 : \tau_1^p, \ldots, x_n = p_n : \tau_n^p\} & \textit{(enregistrement)}
\end{array}
$$

Nous excluons de cette présentation les motifs « ou » (avec la syntaxe « `( | )` » en Caml), ainsi que ceux associés aux tableaux ou aux tuples. Les seules constantes ici sont les entiers n. Notons que les expressions `let`, `fun` et `match` du langage `Typedtree` sont des expressions liant les variables des motifs $p_1 \ldots p_n$ dans les sous-expressions e (pour `let`) ou $e_1 \ldots e_n$ (pour `fun` et `match`).

Les primitives Π sont : $\mathtt{prim}_+$, $\mathtt{prim}_-$, $\mathtt{prim}_*$, $\mathtt{prim}_/$ (opérations arithmétiques), $\mathtt{prim}_{==}$ et $\mathtt{prim}_{!=}$ (comparaisons).

On peut formaliser les types τ de la manière suivante :

$$
\begin{array}{ll}
\tau := \alpha & \textit{(variable de type)} \\
\quad \mid \tau_1 \rightarrow \tau_2 & \textit{(type flèche)} \\
\quad \mid (\tau_1, \ldots, \tau_n)\ def & \textit{(type nommé)}
\end{array}
$$

et pour les définitions de types :

$$\begin{aligned} def := & (\alpha_1,\ldots,\alpha_n)\ pa \\ & \mid (\alpha_1,\ldots,\alpha_n)\ pa = C_1(\tau_1^1,\ldots,\tau_1^{n_1})|\ldots|C_k(\tau_k^1,\ldots,\tau_k^{n_k}) \\ & \mid (\alpha_1,\ldots,\alpha_n)\ pa = \{x_1 : \tau_1,\ldots,x_n : \tau_n\} \end{aligned}$$

qui correspondent respectivement aux types abstraits, variants et enregistrements. Le type *int* et *bool* sont des exemples standard de type nommés (sans paramètre de type). La définition de types pour *int* est abstraite alors qu'on a un type variant pour *bool* : $False()|True()$.

Note : dans l'implantation réelle de Caml les types τ correspondent à une paire `(env,texpr)` formée d'un environnement de typage (défini dans le module `Env`) et d'une expression de type (définie dans le module `Types`). L'utilisation conjointe d'un type et d'un environnement de typage permet de retrouver des informations détaillées sur ce type, en particulier la déclaration d'un type nommé : en effet les expressions de types relatives à un type variant ou enregistrement ne conservent que le nom du type (pleinement qualifié par un chemin de types, certes) sans le détail des caractéristiques de ce type, que l'on peut heureusement retrouver dans l'environnement.

La représentation `Typedlambda`. Le langage `Typedlambda` est formé au dessus de `Lambda` de manière totalement similaire à `Ctypedlambda`. Ici nous décorons chaque nœud d'un arbre de syntaxe `Lambda` au moyen d'une expression de types directement issue de la représentation `Typedtree`. Voici ci-après une formalisation d'un sous-ensemble de `Typedlambda`.

Les expressions typées sont notées $l : \tau$, où l est défini de la manière suivante :

$$\begin{aligned} l := & \ x \\ & \mid let\ x = l_1 : \tau_1\ in\ l_2 : \tau_2 \\ & \mid n \\ & \mid fun\ x_1 : \tau_1 \ldots x_n : \tau_n \to l : \tau \\ & \mid letrec \left\{ \begin{array}{l} x_1 = fun(\ldots) : \tau_1 \\ \vdots \\ x_n = fun(\ldots) : \tau_n \end{array} \right\} in\ l' : \tau' \\ & \mid apply(l : \tau, l_1 : \tau_1,\ldots,l_n : \tau_n) \\ & \mid if\ l : \tau\ then\ l_1 : \tau_1\ else\ l_2 : \tau_2 \\ & \mid switch\ l : \tau\ with \left\{ \begin{array}{l} i_1 \to l_1 : \tau_1 \\ \vdots \\ i_n \to l_n : \tau_n \\ \_ \to l_d : \tau_d \end{array} \right. \\ & \mid catch_i\ l_1 : \tau_1\ with\ x_1 : \tau_1^x \ldots x_n : \tau_n^x \to l_2 : \tau_2 \\ & \mid fail_i(l_1 : \tau_1,\ldots l_n : \tau_n) \\ & \mid prim(\Pi, l_1 : \tau_1,\ldots,l_n : \tau_n) \end{aligned}$$

Les valeurs mutuellement récursives sont restreintes aux valeurs fonctionnelles. Les types τ sont identiques à ceux de `Typedtree`.

Les primitives Π sont $+$, $-$, $*$, $/$, $==$, $!=$, $Field_n$ et $Block_{tag}$: ce sont les mêmes que pour le langage `Ctypedlambda` déjà présenté à la section 2.2.1.2. Remarquons aussi que par rapport à celles de `Typedtree` on a ajouté deux nouvelles primitives pour la manipulation des blocs.

On se permettra d'alléger la notation $l : \tau$ en l quand le contexte le permet.

Difficultés inhérentes à la propagation de types. Propager les types à travers les différentes phases de compilation se concrétise principalement à travers trois types d'intervention sur le code existant (les voici par ordre de difficulté croissante) :

- Le remplacement des langages intermédiaires par leur version enrichie par les types. Du code doit être inséré à de nombreux endroits afin de faire accepter le changement de représentation. Ce n'est pas en général du code compliqué.
- La propagation pure et simple : les nouvelles représentations fournissent un espace permettant de renseigner les types. Il faut maintenant relayer ces informations de types d'une étape à l'autre ou même plus prosaïquement d'une fonction à l'autre au sein du compilateur.
- L'insertion de nouvelles annotations de types : la compilation d'une représentation à la suivante peut introduire beaucoup de code additionnel (l'exemple le plus frappant étant la compilation du filtrage de motifs), le travail pour annoter de code par des types pertinents est alors plus difficile.

En général, les modifications simples sont nombreuses et éparpillées dans le code alors que les transformations complexes sont plus localisées.

2.2.3.2 Propagation sur le noyau du langage et à travers le filtrage de motifs

La propagation des informations de types depuis l'arbre de syntaxe décoré par les types `Typedtree` jusqu'à la représentation `Lambda` est assez simple dans la plupart des cas. Seule la compilation du filtrage de motifs, qui engendre une grande quantité de code à partir de constructions concises au niveau du langage source, a nécessité une intervention plus lourde.

Nous présentons la propagation des types sur les sous-ensembles des langages `Typedtree` et `Lambda` formalisés précédemment. Nous utilisons pour cela deux transformations $\mathfrak{P}$ et $\mathfrak{F}$:

- $\mathfrak{P}$ transforme un terme de `Typedtree` en un terme de `Typedlambda` et utilise $\mathfrak{F}$.
- $\mathfrak{F}$ réalise la compilation du filtrage de motifs et travaille pour cela sur des matrices formées de motifs et de termes de `Typedlambda`.

Compilation du filtrage de motifs. Comme on l'a déjà vu dans l'exemple 2 de la section 2.2.2.1, la compilation du filtrage de motifs est complexe et engendre un code très différent de l'arbre de syntaxe du programme source. Les idées introduites pour optimiser cette partie de la compilation sont exposées dans [35]. L'algorithme utilisé dans le compilateur Caml, qui repose sur l'approche des automates faisant machine arrière (*backtracking automata*), introduit plusieurs améliorations importantes détaillées dans ce même article : la gestion optimisée des motifs « ou » avec liaison de variables et la compilation des backtracks par sauts étiquetés reposant sur une analyse des contextes de filtrage et des structures de cas atteignables.

L'algorithme d'origine travaille sur un argument à filtrer $\vec{x} = (x,x_2,\ldots,x_m)$ (un vecteur de variables) et une matrice de filtrage $P \to L$ dont chaque ligne désigne un cas de filtrage sur $\vec{x}$ et l'action à effectuer pour ce cas :

$$P \to L = \begin{pmatrix} p_1^1 & p_1^2 & \cdots & p_1^m & \to & l_1 \\ p_2^1 & p_2^2 & \cdots & p_2^m & \to & l_2 \\ & & & \vdots & & \\ p_n^1 & p_n^2 & \cdots & p_n^m & \to & l_n \end{pmatrix}$$

Le vecteur argument est formé d'expressions `Lambda` et chaque élément p_i^j est une expression du langage de motifs. Le schéma de compilation prend en argument le couple [10] $(\vec{x},P \to L)$ et retourne l'automate de filtrage optimisé sous forme de code `Lambda`. Des règles détaillées dans l'article permettent de diviser la matrice de filtrage afin d'appliquer récursivement le schéma à des matrices plus simples. Le liant entre les portions de code engendrées pour les sous-cas est constitué de structures de contrôles avancées comme les commandes `catch`, les sauts étiquetés et les structures de `switch`, parmi lesquelles sont insérées les différentes actions de L à appliquer.

Le compilateur OCamIL adapte cette transformation. $\mathfrak{F}$ prend en arguments un vecteur de variables typées $\vec{x} = (x_1 : \tau_1^x, x_2 : \tau_2^x, \ldots, x_m : \tau_m^x)$ et une matrice de filtrage dont les actions sont des termes de `Typedlambda` :

$$P \to L = \begin{pmatrix} p_1^1 & p_1^2 & \cdots & p_1^m & \to & l_1 : \tau_1 \\ p_2^1 & p_2^2 & \cdots & p_2^m & \to & l_2 : \tau_2 \\ & & & \vdots & & \\ p_n^1 & p_n^2 & \cdots & p_n^m & \to & l_n : \tau_n \end{pmatrix}$$

La compilation du filtrage de motifs procède par cas sur la matrice de filtrage et utilise des règles de décomposition des valeurs à filtrer, suivant la classe de motifs structurés à laquelle elles appartiennent : enregistrements, variants, variants polymorphes, tuples... Celles-ci provoquent la génération de primitives d'accès aux valeurs structurées, reposant exclusivement sur la primitive *Field* de lecture d'un élément de bloc Caml. Voici les différents cas de filtrage :

10. Nous simplifions pour cette présentation. En réalité dans sa forme optimisée la compilation du filtrage de motifs manipule des arguments supplémentaires : des informations d'exhaustivité, des contextes de filtrage, et des informations de récupérateurs de sauts atteignables, qui visent à éliminer les redondances et les inefficacités dans le code produit.

Cas de base (vecteur vide) :

$$\mathfrak{F}((), \begin{pmatrix} \to l_1 : \tau_1 \\ \vdots \\ \to l_n : \tau_n \end{pmatrix}) = l_1 : \tau_1$$

Cas variable (la première colonne de la matrice de filtrage est formée uniquement de variables) :

$$\mathfrak{F}((x_1 : \tau_1^x, \ldots, x_m : \tau_m^x), \begin{pmatrix} y_1\ p_1^2 \ldots p_1^m \to l_1 \\ \vdots \\ y_n\ p_n^2 \ldots p_n^m \to l_n \end{pmatrix}) =$$
$$\mathfrak{F}((x_2 : \tau_2^x, \ldots, x_m : \tau_m^x), \begin{pmatrix} p_1^2 \ldots p_1^m \to let\ y_1 = x_1 : \tau_1^x\ in\ l_1 \\ \vdots \\ p_n^2 \ldots p_n^m \to let\ y_n = x_n : \tau_1^x\ in\ l_n \end{pmatrix})$$

Le motif universel « _ » rentre aussi dans ce cas de figure, mais n'a pas besoin d'effectuer de liaison au moyen du *let*.

Cas entier (la première colonne de la matrice de filtrage est formée uniquement d'entiers) :

$$\mathfrak{F}((x_1 \ldots x_m), \begin{pmatrix} i_1\ p_1^2 \ldots p_1^m \to l_1 : \tau_1 \\ \vdots \\ i_n\ p_n^2 \ldots p_n^m \to l_n : \tau_n \end{pmatrix}) =$$
$$\left[\begin{array}{l} if\ x_1 = j_1\ then\ \mathfrak{F}(x_2 \ldots x_m, S(j_1, P \to L))\ else \\ \ldots \\ if\ x_1 = j_k\ then\ \mathfrak{F}(x_2 \ldots x_m, S(j_k, P \to L))\ else\ fail \end{array} \right.$$

où $j_1 \ldots j_k$ est l'ensemble des entiers distincts présents dans la suite $i_1 \ldots i_n$.

La transformation $S(j, P \to L)$ consiste à effacer les lignes dont l'entier de la première colonne est différent de j, et pour les autres à éliminer la première colonne de motifs (par ailleurs les types sont préservés) :

p_i^1	$S(j,P \to L)$
j	$p_i^2 \ldots p_i^m$
$j'(\neq j)$	pas de ligne

Cas enregistrement (la première colonne de la matrice de filtrage est formée uniquement d'enregistrements) :

$$\mathfrak{F}((x_1 : \tau_1^x, \ldots, x_m : \tau_m^x), \begin{pmatrix} \{x^1 = q_1^1 : \tau^1, \ldots, x^r = q_1^r : \tau^r\}\ p_1^2 \ldots p_1^m \to l_1 \\ \vdots \\ \{x^1 = q_n^1 : \tau^1, \ldots, x^r = q_n^r : \tau^r\}\ p_n^2 \ldots p_n^m \to l_n \end{pmatrix}) =$$

$$\left[\begin{array}{l} let\ y_1 = prim(Field_0, x_1 : \tau_1^x) : \tau_1^y\ in \\ \dots \\ let\ y_r = prim(Field_{r-1}, x_1 : \tau_1^x) : \tau_r^y\ in \\ \qquad \mathfrak{F}(y_1 : \tau_1^y, \dots, y_r : \tau_r^y, x_2 : \tau_2^x, \dots, x_m : \tau_m^x, \begin{pmatrix} q_1^1 \dots q_1^r\ p_1^2 \dots p_1^m \rightarrow l_1 \\ \vdots \\ q_n^1 \dots q_n^r\ p_n^2 \dots p_n^m \rightarrow l_n \end{pmatrix}) \end{array}\right.$$

avec pour tout $1 \leq i \leq r$, $\tau_i^y = \tau^i$

Cas variant (la première colonne de la matrice de filtrage est formée uniquement de variants) :

$$\mathfrak{F}((x_1 : \tau_1^x, \dots, x_m : \tau_m^x), \begin{pmatrix} C_1(q_1^1 : \tau_1^1, \dots, q_1^{r_1} : \tau_1^{r_1})\ p_1^2 \dots p_1^m \rightarrow l_1 \\ \vdots \\ C_n(q_n^1 : \tau_n^1, \dots, q_n^{r_n} : \tau_n^{r_n})\ p_n^2 \dots p_n^m \rightarrow l_n \end{pmatrix}) =$$

$$\left[\begin{array}{l} switch\ x_1 : \tau_1^x\ with \\ \left\{\begin{array}{ll} c_1 \rightarrow & \begin{array}{l} let\ y_1 = prim(Field_{0,c_1}, x_1 : \tau_1^x) : \tau_{c_1,1}^y\ in \\ \dots \\ let\ y_{a_1} = prim(Field_{a_1-1,c_1}, x_1 : \tau_1^x) : \tau_{c_1,a_1}^y\ in \\ \quad \mathfrak{F}((y_1 : \tau_{c_1,1}^y, \dots, y_{a_1} : \tau_{c_1,a_1}^y, x_2 : \tau_2^x, \dots, x_m : \tau_m^x), S(c_1, P \rightarrow L)) \end{array} \\ \vdots & \\ c_k \rightarrow & \begin{array}{l} let\ y_1 = prim(Field_{0,c_k}, x_1 : \tau_1^x) : \tau_{c_k,1}^y\ in \\ \dots \\ let\ y_{a_k} = prim(Field_{a_k-1,c_k}, x_1 : \tau_1^x) : \tau_{c_k,a_k}^y\ in \\ \quad \mathfrak{F}((y_1 : \tau_{c_k,1}^y, \dots, y_{a_k} : \tau_{c_k,a_k}^y, x_2 : \tau_2^x, \dots, x_m : \tau_m^x), S(c_k, P \rightarrow L)) \end{array} \\ \_ \rightarrow fail & \end{array}\right. \end{array}\right.$$

où $c_1 \dots c_k$ sont les tags distincts associés aux constructeurs présents sur la première colonne, et $a_1 \dots a_k$ sont les arités de ces constructeurs. De plus $\forall i,j\ \tau_{i,j}^y = \tau_l^j$ avec l tel que $tag(C_l) = c_i$.

On peut remarquer que les primitives *Field* sont utilisées sous leur forme étendue, gardant la trace du tag du constructeur.

La transformation $S(c, P \rightarrow L)$ consiste à effacer les lignes dont le constructeur de la première colonne est différent de c, et pour les autres à réécrire le motif du constructeur en expansant au moyen de ses arguments (par ailleurs les types sont préservés) :

p_i^1	$S(c, P \rightarrow L)$
$c(q_i^1, \dots, q_i^a)$	$q_i^1 \dots q_i^a\ p_i^2 \dots p_i^m$
$c'(q_i^1, \dots, q_i^{a'})\ (c' \neq c)$	pas de ligne

Notons que nous utilisons une construction *switch* qui opère sur les tags des constructeurs. On suppose ici pour simplifier que l'on a une représentation homogène des constructeurs constants et non constants comme suggérée dans la section 2.2.2.2 pour remédier au problème de retypage des types variant.

Cas division :

Ce cas consiste à diviser verticalement la matrice de filtrage afin de tomber sur l'un des cas précédents.

$$\mathfrak{F}((x_1 \dots x_m), \begin{pmatrix} p_1^1 \dots p_1^m \rightarrow l_1 \\ \vdots \\ p_n^1 \dots p_n^m \rightarrow l_n \end{pmatrix}) =$$
$$\left[\begin{array}{l} catch\ \mathfrak{F}((x_1 \dots x_m), \begin{pmatrix} p_1^1 \dots p_1^m \rightarrow l_1 \\ \vdots \\ p_k^1 \dots p_k^m \rightarrow l_k \end{pmatrix}) \\ \qquad with\ \mathfrak{F}((x_1 \dots x_m), \begin{pmatrix} p_{k+1}^1 \dots p_{k+1}^m \rightarrow l_{k+1} \\ \vdots \\ p_n^1 \dots p_n^m \rightarrow l_n \end{pmatrix}) \end{array} \right.$$

où k est le plus grand tel que l'un des cas précédents s'applique sur la matrice supérieure. Nous avons présenté ici ce cas sous sa forme la plus simple. Le compilateur Caml optimise ce cas en procédant à des permutations de lignes respectant la sémantique du filtrage, et dont le but est de former des sous-matrices plus grosses.

Il est facile de retrouver par recoupement les types devant décorer les structures de contrôle introduites par la transformation (les $catch$, les $fail$, les if et les $switch$). Le point crucial est de bien typer les primitives $Field$ et il faut pour cela analyser les types filtrés afin d'en extraire les types internes. Heureusement, les motifs eux-mêmes contiennent des informations de types (y compris dans le compilateur Caml d'origine) car celles-ci sont exploitées par l'algorithme de filtrage : par exemple lorsqu'on filtre sur des constructeurs d'un type variant, la connaissance précise de la définition de ce type permet de déterminer si le filtrage est exhaustif. De même la génération des primitives d'accès adéquates suppose de connaître la représentation des valeurs filtrées dans l'environnement d'exécution Caml, ce qui n'est pas donné par la syntaxe du filtrage mais par la définition du type filtré. Des détails supplémentaires sur l'implantation réelle sont donnés un peu plus loin.

Transformation du noyau du langage. La transformation $\mathfrak{P}$ est assez élémentaire et repose sur $\mathfrak{F}$. Elle est définie comme suit :

- $\mathfrak{P}(\mathtt{x} : \tau) = x : \tau$

- $\mathfrak{P}(\mathtt{let} \left\{ \begin{array}{l} p_1 = \mathtt{e_1} : \tau_1 \\ \vdots \\ p_n = \mathtt{e_n} : \tau_n \end{array} \right\} \mathtt{in}\ \mathtt{e} : \tau) =$
$$\mathfrak{P}(\mathtt{match}\ \mathtt{e_1} : \tau_1\ \mathtt{with}\ p_1 \rightarrow \dots \rightarrow \mathtt{match}\ \mathtt{e_n} : \tau_n\ \mathtt{with}\ p_n \rightarrow \mathtt{e} : \tau)$$

• $\mathfrak{P}(\mathtt{n} : \tau) = n : \tau$

• $\mathfrak{P}(\{\mathtt{x_1} = \mathtt{e_1} : \tau_1, \ldots, \mathtt{x_n} = \mathtt{e_n} : \tau_n\} : \tau) = prim(Block_0, \mathfrak{P}(\mathtt{e_1} : \tau_1), \ldots, \mathfrak{P}(\mathtt{e_n} : \tau_n)) : \tau$

• $\mathfrak{P}(\mathtt{C}(\mathtt{e_1} : \tau_1, \ldots, \mathtt{e_n} : \tau_n) : \tau) = prim(Block_{tag(C)}, \mathfrak{P}(\mathtt{e_1} : \tau_1), \ldots, \mathfrak{P}(\mathtt{e_n} : \tau_n)) : \tau$

• $\mathfrak{P}(\mathtt{fun} \left\{ \begin{array}{l} p_1 \rightarrow \mathtt{e_1} \\ \vdots \\ p_n \rightarrow \mathtt{e_n} \end{array} \right\} : \tau \rightarrow \tau') =$

$$(fun\ x_1 : \tau \rightarrow \mathfrak{P}(\mathtt{match}\ \mathtt{x_1} : \tau\ \mathtt{with} \left\{ \begin{array}{l} p_1 \rightarrow \mathtt{e_1} \\ \vdots \\ p_n \rightarrow \mathtt{e_n} \end{array} \right\} : \tau')) : \tau \rightarrow \tau'$$

• $\mathfrak{P}(\mathtt{let}^* \left\{ \begin{array}{l} x_1 = \mathtt{fun}\ \{\ldots\} \\ \vdots \\ x_n = \mathtt{fun}\ \{\ldots\} \end{array} \right\} \mathtt{in}\ \mathtt{e} : \tau) = letrec \left\{ \begin{array}{l} x_1 = \mathfrak{P}(\mathtt{fun}\ \{\ldots\}) \\ \vdots \\ x_n = \mathfrak{P}(\mathtt{fun}\ \{\ldots\}) \end{array} \right\} in\ \mathfrak{P}(\mathtt{e} : \tau)$

• $\mathfrak{P}(\mathtt{apply}(\mathtt{e} : \tau, \mathtt{e_1} : \tau_1, \ldots, \mathtt{e_n} : \tau_n) : \tau') = apply(\mathfrak{P}(\mathtt{e} : \tau), \mathfrak{P}(\mathtt{e_1} : \tau_1), \ldots, \mathfrak{P}(\mathtt{e_n} : \tau_n)) : \tau'$

• $\mathfrak{P}(\mathtt{if}\ \mathtt{e} : \tau\ \mathtt{then}\ \mathtt{e_1} : \tau_1\ \mathtt{else}\ \mathtt{e_2} : \tau_2) = if\ \mathfrak{P}(\mathtt{e} : \tau)\ then\ \mathfrak{P}(\mathtt{e_1} : \tau_1)\ else\ \mathfrak{P}(\mathtt{e_2} : \tau_2)$

• $\mathfrak{P}(\mathtt{match}\ \mathtt{e} : \tau\ \mathtt{with} \left\{ \begin{array}{l} p_1 \rightarrow \mathtt{e_1} \\ \vdots \\ p_n \rightarrow \mathtt{e_n} \end{array} \right\} : \tau') = let\ x = \mathfrak{P}(\mathtt{e} : \tau)\ in\ \mathfrak{F}((x : \tau), P \rightarrow L) : \tau'$

avec $P \rightarrow L = \left(\begin{array}{l} p_1 \rightarrow \mathfrak{P}(\mathtt{e_1} : \tau') \\ \vdots \\ p_n \rightarrow \mathfrak{P}(\mathtt{e_n} : \tau') \end{array} \right)$ (matrice de filtrage à une seule colonne).

• $\mathfrak{P}(\mathtt{prim}_\Pi : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau_r) = fun\ x_1 : \tau_1 \ldots x_n : \tau_n \rightarrow prim(\Pi, x_1, \ldots, x_n) : \tau_r$

Commentaires sur l'implantation réelle. La présentation formelle ci-dessus rend compte de l'essentiel de la transformation, mais il est utile de la compléter par quelques observations sur la transformation réelle.

Utilisation des structures *catch* et *fail* enrichies : outre l'optimisation de la règle de division mentionnée plus haut, l'algorithme décrit dans [35] utilise des structures *catch* et *fail* étiquetées par des entiers et permettant la transmission de valeurs :

L'algorithme décrit plus haut introduit des emboîtements de structures *catch*, et il arrive souvent en pratique qu'un échec dans une structure *catch* intérieure conduit automatiquement à un échec dans la structure *catch* extérieure. L'algorithme optimisé réussit à détecter ces cas statiquement et génère un saut court-circuitant les structures *catch* intermédiaires : c'est la raison des étiquettes entières.

La compilation des motifs « ou » comportant des variables n'est pas décrite ici. Il s'agit de traiter des motifs de la forme (l'exemple qui suit est adapté de [35]) :

```
let car2 list = match list with
  | Nil -> -1
  | (One x | Cons (x,_)) -> x * 2
```

La deuxième action sera traitée par le cas d'échec d'une seule structure *catch* : celle-ci doit être capable de récupérer la valeur de `x` qui est transmise par les instructions $fail$ enrichies (voir par exemple la définition de `Clambda` à la section 2.1.1.4).

Cas du filtrage non-exhaustif : la compilation de la construction `match` présentée ci-dessus ne tient pas compte du filtrage non-exhaustif. La règle de compilation s'écrit en réalité :

$$\mathfrak{P}(\texttt{match e} : \tau \texttt{ with} \left\{ \begin{array}{l} p_1 \rightarrow \texttt{e}_1 \\ \vdots \\ p_n \rightarrow \texttt{e}_\texttt{n} \end{array} \right\} : \tau') =$$
$$let\ x = \mathfrak{P}(\texttt{e} : \tau)\ in\ catch\ \mathfrak{F}((x : \tau), P \rightarrow L)\ with\ raise("MatchFailure") : \tau'$$

Cependant nous avons exclu les exceptions de notre sous-langage formalisé.

Propagation des informations de types manquantes : l'information récupérée dans les motifs de filtrage des variants et des enregistrements, nécessaire pour typer les primitives $Field$ engendrées n'est en réalité pas suffisamment précise.

En effet les types contenus dans les motifs ne sont pas instanciés (car ce n'est pas utile pour leur emploi dans le compilateur Caml d'origine). Par exemple si on considère l'expression suivante :

```
match l with [] -> 0 | x::_ -> x * 2
```

L'argument `l` est porteur du type exact *int list* mais le motif `x::_` donnera les types non instanciés α et α *list* à ses composants. L'algorithme de propagation des types de OCamIL procède à des recoupements entre les types donnés pour le vecteur argument du filtrage et les types des motifs afin de reconstituer les types exacts.

Nous donnons dans la suite une idée de la nature des transformations à apporter au code Caml d'origine afin de propager l'information de types, en particulier pour tenir compte de l'observation précédente.

Le module `Matching` du code source de Caml contient plusieurs jeux de fonctions sur chaque sorte de motif, avec entre autres :

- `get_args_*` et `matcher_*` qui sont responsables de l'extraction du sous-motif : par exemple un motif $p = Cons(p_1,p_2)$ élément d'une matrice de filtrage $P \rightarrow L$ peut engendrer une nouvelle matrice filtrant les sous-motifs de p et construite autour de p_1 et p_2.

– `make_*_matching` qui effectue le passage à la compilation aux sous-matrices, en combinant les fonctions précédentes, générant le code `Clambda` qui destructure l'argument $\vec{x} = (x_1,x_2,\ldots,x_n)$ du filtrage à l'aide d'accesseurs et en calculant la structure de gestion d'échec local du filtrage. Si par exemple le motif $p = Cons(p_1,p_2)$ ci-dessus est amené à filtrer x_2 alors le nouvel argument sera de la forme $(x_1,prim(Field_0,x_2),prim(Field_1,x_2),\ldots,x_n)$.

La version modifiée du module `Matching` contenue dans OCamIL doit insérer les bons types autour des accesseurs compilés. Nous illustrons ces modifications dans le cas des enregistrements. L'exemple suivant est directement tiré du code source de Objective Caml :

```
let make_record_matching all_labels def = function
    [] -> fatal_error "Matching.make_record_matching"
  | ((arg, mut) :: argl) ->
      let rec make_args pos =
        if pos >= Array.length all_labels then argl else begin
          let lbl = all_labels.(pos) in
          let access =
            match lbl.lbl_repres with
              Record_regular -> Pfield lbl.lbl_pos
            | Record_float -> Pfloatfield lbl.lbl_pos in
          let str =
            match lbl.lbl_mut with
              Immutable -> Alias
            | Mutable -> StrictOpt in
          (Lprim(access, [arg]), str) :: make_args(pos + 1)
        end in
      let nfields = Array.length all_labels in
      let def= make_default (matcher_record nfields) def in
      {cases = []; args = make_args 0 ; default = def}
```

Les arguments `all_labels` et `def` sont respectivement : 1) la description des champs de l'enregistrement filtré (un tableau de valeurs décrivant la structure typée des champs de l'enregistrement, sous forme non instanciée) et 2) une représentation des opérations à faire en cas de défaut (ce que l'article [35] appelle les *reachable trap handlers*). Le dernier argument représente le vecteur argument du matching $\vec{x} = (x_1 \ldots)$ dont c'est la première composante qui est ici filtrée sur un motif d'enregistrement. On peut voir le travail de la fonction locale `make_args` qui remplace x_1 par la série des accès aux champs de x_1. Nous ne détaillons pas la mise à jour de `def` par la fonction `matcher_record`.

Voici la même fonction dans les sources de OCamIL (les différences apparaissent en gris) :

```
let make_record_matching all_labels (pat_type,pat_env) def = function
    [] -> fatal_error "Matching.make_record_matching"
  | ((arg, mut) :: argl) ->
```

```
let rec make_args pos =
  if pos >= Array.length all_labels then argl else begin
    let lbl = all_labels.(pos) in
      let (lblargs,recres) = Ctype.instance_parameterized_type
                                      [lbl.lbl_arg] lbl.lbl_res in
      Ctype.unify pat_env recres pat_type;
      let lblarg = List.hd lblargs in (* there is exactly one *)
      let access =
        match lbl.lbl_repres with
          Record_regular -> Pfield lbl.lbl_pos
        | Record_float -> Pfloatfield lbl.lbl_pos in
      let str =
        match lbl.lbl_mut with
          Immutable -> Alias
        | Mutable -> StrictOpt in
      let item_type = build_type_annotation lblarg pat_env in
      (build_term (TypLprim(access, [arg])) item_type,
      str) :: make_args(pos + 1)
    end in
  let nfields = Array.length all_labels in
  let def= make_default (matcher_record nfields) def in
  {cases = []; args = make_args 0 ; default = def}
```

Dans le cas de types enregistrement paramétrés, l'information contenue dans l'argument `all_labels` est générique, quand bien-même les motifs travaillent sur une instanciation de type enregistrement. Afin de donner le type le plus précis possible au résultat de la primitive d'accès sur chaque champ, nous prenons une instance du schéma de types donné pour le type enregistrement et nous l'unifions au type lu directement sur le motif (propagé grâce aux arguments supplémentaires `(pat_type,pat_env)`). Nous récupérons une version instanciée du type de chaque étiquette que nous accolons à la primitive d'accès.

Les opérations sur les types Caml sont directement menées à bien en exploitant à la façon d'une API les modules du compilateur qui servent normalement d'adjoints au typeur, ici le module `Ctype`. C'est parce que dans les premières phases de la propagation de types nous sommes conduits à effectuer ce genre d'opérations que nous gardons la représentation originelle des types Caml. Une fois tous les types complétés et confrontés au sein des représentations intermédiaires, nous pouvons figer ces types dans une représentation plus simple : c'est ce qui se produit avec `Ctypedlambda` et `typeinfo`.

Les types variant sont traités de manière similaire : nous procédons aussi à une unification entre un descripteur de types donné par l'algorithme de filtrage et le type effectivement propagé ; il suffit juste de se servir de l'information de tag de bloc pour déterminer les types arguments de chaque constructeur.

Modules et foncteurs. Objective Caml représente les modules comme des tableaux de valeurs (y compris des valeurs fonctionnelles) et sont ainsi compilés vers des blocs. Les foncteurs de premier ordre sont des fonctions des modules dans les modules, les foncteurs de deuxième ordre sont des fonctions des modules et des foncteurs de premier ordre dans les modules et ainsi de suite. Un foncteur quelconque est donc une fonction dont le type flèche a une structure arborescente quelconque mais dont les feuilles pourront être typées par block.

Au niveau de la propagation des types, l'implantation de OCamIL choisit de rester générique sur les modules et les foncteurs et les type de la manière suivante :
- Les expressions de `Lambda` engendrées à partir du langage de modules reçoivent le type $\alpha\ array$.
- Les expressions fonctorielles construites sur deux sous-expressions de types τ_1 et τ_2 reçoivent le type $\tau_1 \rightarrow \tau_2$.

Nous verrons à la section 3.1.4.2 du chapitre suivant que les limitations imposées par le système de types CTS sur les implantations possibles des modules et des foncteurs expliquent que nous n'ayons pas cherché davantage de précision sur les types propagés.

2.2.3.3 Propagation des types entre `Lambda` et `Clambda`.

Le module `Closure` est chargé de transformer la représentation `Lambda` en représentation `CLambda` par l'explicitation des blocs fermetures. Dans l'optique de la propagation de types, il est simple d'étendre cette transformation au couple `Typedlambda`/`Ctypedlambda`.

Durant cette transformation les types Caml sont convertis vers la grammaire `typeinfo`, plus simple, et servant d'interface commune aux schémas de propagation et de reconstruction de types.

On peut définir cette transformation sur les types Θ de la façon suivante :

- $\Theta(\alpha) = \mathsf{obj}$
- $\Theta(\tau_1 \rightarrow \tau_2) = \Theta(\tau_1) \rightarrow \Theta(\tau_2)$
- $\Theta((\tau_1, \ldots, \tau_n)\ def)$ dépend de la déclaration de types def :
  - types abstraits $(def : (\alpha_1, \ldots, \alpha_n)\ pa)$: on isole d'abord le cas particulier $\Theta(int) = \mathsf{int}$ et sinon $\Theta((\tau_1, \ldots, \tau_n)\ def) = \mathsf{obj}$.
  - types variant $(def : (\alpha_1, \ldots, \alpha_n)\ pa = C_1(\tau_1^1, \ldots, \tau_1^{n_1}) | \ldots | C_k(\tau_k^1, \ldots, \tau_k^{n_k}))$: on a le cas $\Theta(bool) = \mathsf{bool}$ et sinon $\Theta((\tau_1, \ldots, \tau_n)\ def) = \mathsf{variant}_{\mathsf{pa},\mathsf{dv}}$ avec $\mathsf{dv} = (\mathsf{C}_1 : (\Theta(\tau_1^1), \ldots, \Theta(\tau_1^{n_1}) | \ldots | \mathsf{C}_\mathsf{k} : (\Theta(\tau_k^1), \ldots, \Theta(\tau_k^{n_k})))$.
  - types enregistrement $(def : (\alpha_1, \ldots, \alpha_n)\ pa = \{x_1 : \tau_1, \ldots, x_n : \tau_n\})$: $\Theta((\tau_1, \ldots, \tau_n)\ def) = \mathsf{variant}_{\mathsf{pa},\mathsf{dr}}$ avec $\mathsf{dr} = (\mathsf{x}_1 : \Theta(\tau_1), \ldots, \mathsf{x}_\mathsf{n} : \Theta(\tau_n))$.

La transformation Θ présentée ci-dessus sur un noyau minimal s'étend naturellement à l'ensemble des types Caml.

La transformation des termes `Typedlambda` à `Ctypedlambda` est notée $\mathfrak{C}_E$. Elle utilise un environnement formé d'une paire (f,d) :

- f sert à remplacer les variables libres par des primitives d'accès à l'environnement de la fermeture courante. On notera $f(x) = \mathbf{u}\colon \mathtt{t}$ où $\mathbf{u}$ est le terme de `Ctypedlambda` réalisant l'accès.
- d est utilisé pour générer des applications de fonctions directes en mémorisant des associations entre variables et étiquettes de fonctions. On notera $d(x) = (\mathbf{L},e)$ où $\mathbf{L}$ est l'étiquette d'une fonction liée à la variable x et e est un booléen qui est vrai lorsque la fonction attend un argument supplémentaire pour gérer son environnement.

De nombreux cas de la transformation sont élémentaires :

- $\mathfrak{C}_{f,d}(n : \tau) = \mathbf{n}\colon \Theta(\tau)$

- $\mathfrak{C}_{f,d}(if\ l : \tau\ then\ l_1 : \tau_1\ else\ l_2 : \tau_2) = \mathbf{if}\ \mathfrak{C}_{f,d}(l : \tau)\ \mathbf{then}\ \mathfrak{C}_{f,d}(l : \tau_1)\ \mathbf{else}\ \mathfrak{C}_{f,d}(l : \tau_2)$

- $\mathfrak{C}_{f,d}(switch\ l : \tau\ with \begin{cases} i_1 \to l_1 : \tau_1 \\ \vdots \\ i_n \to l_n : \tau_n \\ \_ \to l_d : \tau_d \end{cases}) = \mathbf{switch}\ \mathfrak{C}_{f,d}(l : \tau)\ \mathbf{with} \begin{cases} \mathbf{i_1} \to \mathfrak{C}_{f,d}(l_1 : \tau_1) \\ \vdots \\ \mathbf{i_n} \to \mathfrak{C}_{f,d}(l_n : \tau_n) \\ \_ \to \mathfrak{C}_{f,d}(l_d : \tau_d) \end{cases}$

- $\mathfrak{C}_{f,d}(catch_i\ l_1 : \tau_1\ with\ x_1 : \tau_1^x \ldots x_n : \tau_n^x \to l_2 : \tau_2) =$
$$\mathbf{catch_i}\ \mathfrak{C}_{f,d}(l_1 : \tau_1)\ \mathbf{with}\ \mathbf{x_1}\colon \Theta(\tau_1^{\mathsf{x}}), \ldots, \mathbf{x_n}\colon \Theta(\tau_{\mathsf{n}}^{\mathsf{x}}) \to \mathfrak{C}_{f,d}(l_2 : \tau_2)$$

- $\mathfrak{C}_{f,d}(fail_i(l_1 : \tau_1, \ldots, l_n : \tau_n)) = \mathbf{fail_i}(\mathfrak{C}_{f,d}(l_1 : \tau_1), \ldots, \mathfrak{C}_{f,d}(l_n : \tau_n))$

- $\mathfrak{C}_{f,d}(prim(\Pi, l_1 : \tau_1, \ldots, l_n : \tau_n) : \tau) = \mathbf{prim}(\mathbf{\Pi}, \mathfrak{C}_{f,d}(l_1 : \tau_1), \ldots, \mathfrak{C}_{f,d}(l_n : \tau_n))\colon \Theta(\tau)$

Les autres cas font intervenir l'environnement :

- $\mathfrak{C}_{f,d}(x : \tau) = \mathbf{u}\colon \mathtt{t}$ si $f(x) = \mathbf{u}\colon \mathtt{t}$

- $\mathfrak{C}_{f,d}(x : \tau) = \mathbf{x} :\Theta(\tau)$ lorsque $f(x)$ n'est pas défini.

- $\mathfrak{C}_{f,d}((let\ x = l_1 : \tau_1\ in\ l_2 : \tau_2) : \tau) = \mathbf{let}\ \mathbf{x} = \mathfrak{C}_{f,d}(l_1 : \tau_1)\ \mathbf{in}\ \mathfrak{C}_{f,d'}(l_2 : \tau_2)\colon \Theta(\tau)$
avec :
  - d' étend d en posant $d'(x) = \mathbf{L}$ dans le cas où $\mathfrak{C}_{f,d}(l_1 : \tau_1) = \mathbf{closure}(\mathbf{L}, \ldots)$
  - $d' = d$ sinon

- $\mathfrak{C}_{f,d}(fun\ x_1 : \tau_1 \ldots x_n : \tau_n \to l : \tau) =$
$$\left.\begin{array}{c}\mathbf{closure}\left(\begin{array}{c}\mathbf{L},\mathbf{a},\mathbf{x_1}\colon \Theta(\tau_1)\ldots \mathbf{x_n}\colon \Theta(\tau_{\mathsf{n}}),\mathbf{env}^*\colon \mathsf{clos}, \\ \mathfrak{C}_{f',d}(l : \tau)\end{array}\right) \\ (\mathfrak{C}_{f,d}(y_1 : \tau_1^y), \ldots, \mathfrak{C}_{f,d}(y_m : \tau_m^y))\end{array}\right\} : \Theta(\tau_1 \to \ldots \to \tau_{\mathsf{n}} \to \tau)$$

où

- **L** est une étiquette unique attribuée à la fonction (on suppose l'existence d'un tel générateur d'étiquettes).
- $y_1,\ldots,y_m$ sont les variables libres dans le corps l de la fonction.
- f' étend f en ajoutant les associations : $y_i \rightarrow \mathbf{prim}(\mathbf{Field_{p_i}},\mathbf{env}\text{: clos})$ pour $1 \leq i \leq m$ (chaque $\mathbf{p_i}$ désigne la position du i-ème champ de l'environnement de la fermeture, cette position est calculée de par la connaissance exacte de la représentation des fermetures dans l'environnement d'exécution Caml).
- L'argument supplémentaire **env** n'est présent que si les variables libres existent [11].

$$\bullet\ \mathfrak{C}_{f,d}(letrec \left\{ \begin{array}{l} x_1^f = fun\ x_1^1 : \tau_1^1 \ldots x_1^{n_1} : \tau_1^{n_1} \rightarrow l_1 : \tau_1 \\ \vdots \\ x_p^f = fun\ x_p^1 : \tau_p^1 \ldots x_p^{n_p} : \tau_p^{n_p} \rightarrow l_p : \tau_p \end{array} \right\} in\ l : \tau) =$$

$$\mathbf{let\ s} = \begin{array}{l} \mathbf{closure} \left(\begin{array}{l} \mathbf{L_1},\mathbf{a_1},\mathbf{x_1^1}\text{: }\Theta(\tau_1^1)\ldots\mathbf{x_1^{n_1}}\text{: }\Theta(\tau_1^{\mathsf{n_1}}),\mathbf{e}^*\text{: sclos},\mathfrak{C}_{f'_1,d}(l_1 : \tau_1) \\ \vdots \\ \mathbf{L_p},\mathbf{a_p},\mathbf{x_p^1}\text{: }\Theta(\tau_{\mathsf{p}}^1)\ldots\mathbf{x_p^{n_p}}\text{: }\Theta(\tau_{\mathsf{p}}^{\mathsf{n_p}}),\mathbf{e}^*\text{: sclos},\mathfrak{C}_{f'_p,d}(l_p : \tau_p) \end{array} \right) \\ (\mathfrak{C}_{f,d}(y_1 : \tau_1^y),\ldots,\mathfrak{C}_{f,d}(y_m : \tau_m^y)) \end{array} \text{: sclos}\ \mathbf{in}$$

$$\mathbf{let\ x_1^f} = \mathbf{offset_{o_1}}(\mathbf{s})\text{: }\Theta(\tau_1^1 \rightarrow \ldots \rightarrow \tau_1^{\mathsf{n_1}} \rightarrow \tau_1)\ \mathbf{in}$$
$$\ldots$$
$$\mathbf{let\ x_p^f} = \mathbf{offset_{o_p}}(\mathbf{s})\text{: }\Theta(\tau_{\mathsf{p}}^1 \rightarrow \ldots \rightarrow \tau_{\mathsf{p}}^{\mathsf{n_p}} \rightarrow \tau_{\mathsf{p}})\ \mathbf{in}\quad \mathfrak{C}_{f,d}(l : \tau)$$

où

- $\mathbf{L_1},\ldots,\mathbf{L_p}$ sont des étiquettes uniques attribuées aux fonctions.
- $y_1,\ldots,y_m$ est l'ensemble des variables qui sont libres dans le corps d'au moins une fonction (mais sans inclure $x_1^f,\ldots,x_p^f$).
- pour tout $1 \leq i \leq m$, $\mathbf{p_i}$ désigne la position du i-ème champ de l'environnement de la fermeture partagée.
- pour tout $1 \leq i \leq p$, $\mathbf{o_i}$ désigne la position du i-ème pointeur de fonction dans la fermeture partagée.
- pour tout $1 \leq j \leq p$, f'_j est formé à partir de f en ajoutant les associations :
  - $y_i \rightarrow \mathbf{prim}(\mathbf{Field_{p_i - o_j}},\mathbf{e}\text{: sclos})$ pour $1 \leq i \leq m$
  - $x_i^f \rightarrow \mathbf{offset_{o_i - o_j}}(\mathbf{e})\text{: }\Theta(\tau_{\mathsf{i}}^1 \rightarrow \ldots \rightarrow \tau_{\mathsf{i}}^{\mathsf{n_i}} \rightarrow \tau_{\mathsf{i}})$ pour $1 \leq i \leq p$
- L'argument supplémentaire **e** n'est présent que si les variables libres existent [11].

$$\bullet\ \mathfrak{C}_{f,d}(apply(l : \tau,l_1 : \tau_1,\ldots,l_n : \tau_n)) = \textit{(cas général)}$$
$$\mathbf{Gapply}(\mathfrak{C}_{f,d}(l : \tau),\mathfrak{C}_{f,d}(l_1 : \tau_1),\ldots,\mathfrak{C}_{f,d}(l_n : \tau_n))$$

11. En réalité il est un cas où les variables libres existent mais l'argument supplémentaire n'est pas engendré par le compilateur Caml : c'est lorsque ces variables sont toutes des fonctions qui ne sont utilisées que dans le cadre d'applications directes qui à leur tour ne nécessitent pas d'environnement (car alors seules les étiquettes de ces fonctions sont utilisées et les variables disparaissent).

sauf si $a \leq n$ et que l'on a soit :

- $\mathfrak{C}_{f,d}(l : \tau) = \mathbf{closure}(\mathbf{L},\mathbf{a},\ldots)$ ou bien :
- l est une variable x et $d(x)$ est défini et vaut $d(x) = (\mathbf{L},e)$.

Dans ces cas on a :

- $(a = n)$ $\mathfrak{C}_{f,d}(apply(l : \tau,l_1 : \tau_1,\ldots,l_n : \tau_n)) =$
$$\begin{cases} \mathbf{Dapply}(\mathbf{L},\mathfrak{C}_{f,d}(l_1 : \tau_1),\ldots,\mathfrak{C}_{f,d}(l_n : \tau_n)) & \text{si } e \text{ est faux} \\ \mathbf{Dapply}(\mathbf{L},\mathfrak{C}_{f,d}(l_1 : \tau_1),\ldots,\mathfrak{C}_{f,d}(l_n : \tau_n),\mathfrak{C}_{f,d}(l : \tau)) & \text{si } e \text{ est vrai} \end{cases}$$
- $(a < n)$

$$\mathfrak{C}_{f,d}(apply(l : \tau,l_1 : \tau_1,\ldots,l_n : \tau_n)) = \mathbf{Gapply}(\mathbf{u},\mathfrak{C}_{f,d}(l_{a+1} : \tau_{a+1}),\ldots,\mathfrak{C}_{f,d}(l_n : \tau_n))$$

où

$$\mathbf{u} = \begin{cases} \mathbf{Dapply}(\mathbf{L},\mathfrak{C}_{f,d}(l_1 : \tau_1),\ldots,\mathfrak{C}_{f,d}(l_a : \tau_a)) & \text{si } e \text{ est faux} \\ \mathbf{Dapply}(\mathbf{L},\mathfrak{C}_{f,d}(l_1 : \tau_1),\ldots,\mathfrak{C}_{f,d}(l_a : \tau_a),\mathfrak{C}_{f,d}(l : \tau)) & \text{si } e \text{ est vrai} \end{cases}$$

2.2.4 Gestion des définitions et des références de types nommés

Récupération des définitions de types et chemins de types. Les types nommés définis dans une unité de compilation ne trouvent pas écho dans les représentations intermédiaires : d'une part il n'y a pas de code engendré pour les définitions de types et d'autre part les types nommés qui décorent les langages intermédiaires introduits par OCamIL ne sont pas en parfaire correspondance avec les types définis (on peut avoir des références vers des types définis dans une unité de compilation externe et inversement les types définis dans une unité de compilation peuvent ne pas être utilisés dans celle-ci).

Le compilateur OCamIL doit cependant connaître les types variants et enregistrements définis dans une unité de compilation afin de pouvoir engendrer les classes CTS qui vont leur servir d'implantation.

Il nous a fallu pour cela intervenir sur le module `Typemod` du typeur Caml qui s'occupe des modules et signatures de modules. Celui-ci traite tous les cas de la grammaire de signatures de modules, dont la déclaration de types fait partie. Lorsqu'une définition de variants et d'enregistrements est rencontrée, nous stockons la déclaration dans une table `camil_typedecls`. L'information complète qu'il faut récupérer pour chaque type comporte :

- la définition elle-même,
- l'espace de nom absolu dans lequel le type est déclaré.

Cette dernière information est nécessaire pour ne pas confondre des types homonymes définis dans des modules différents. Nous la constituons en gardant la trace des modules ouverts ou refermés dans une variable globale du même module `Typemod` : pour chaque construction `module M = struct S end` analysée, la structure `S` est traitée dans un contexte où le chemin courant est rallongé par le nom de module `M`.

La table `camil_typedecls` implante l'association entre les chemins de types absolus et leur définition précise.

Chemins de types dans les références. La transformation Θ qui fait passer des expressions de types Caml à la grammaire `typeinfo` est dans la pratique plus complexe sur les types nommés que ce qui a été présenté plus haut.

Le problème vient du fait que les références aux types nommés qui se trouvent dans l'arbre de syntaxe typé `Typedtree` ne contiennent pas directement le chemin de types absolu (et il a donc ambiguïté pour des types homonymes déclarés dans des modules distincts).

Les références de types contenues dans `Typedtree` sont de deux types :

- Externe : la référence est faite à un type défini dans un autre module physique (un fichier d'implantation ou d'interface différent de l'unité de compilation courante). Dans ce cas le chemin de types est absolu et identifie sans ambiguïté le type référencé.
- Interne : la référence concerne un type défini dans la même unité de compilation. Il n'y a alors pas moyen de savoir si le chemin de types est exprimé de manière absolue par rapport à la racine de l'unité de compilation ou bien relative au site porteur de la référence.

Le problème est donc de pouvoir identifier de manière univoque les types référencés dans le cas interne afin de les transformer en références de classe CTS appropriées.

Nous n'avons pas trouvé le moyen d'exploiter le contexte de typage pour récupérer l'information manquante. Pour y remédier nous avons enrichi les annotations de types Caml dans le langage `Typedlambda` en spécifiant le contexte d'imbrication de modules (c'est-à-dire l'espace de noms absolu) qui est actif pour chaque nœud de l'arbre `Typedlambda`. Ces informations sont recoupées avec les chemins stockés dans la table `camil_typedecls` (voir plus haut) des types déclarés dans l'unité de compilation courante.

L'espace de noms courant est maintenu dans une variable globale du module `Translmod`, qui au sein de la transformation de `TypedTree` vers `Lambda`, se charge spécifiquement du langage de modules. Lors de la traversée de sous-modules, le chemin contenu dans cette variable globale est allongé ou raccourci. Toute génération d'expression `Typedlambda` utilise cette variable globale pour insérer l'espace de noms actuel dans l'annotation de type enrichie.

La phase de recoupement des informations consiste à identifier la référence de type locale dans l'arbre des types déclarés, stocké dans la table `camil_typedecls`, en commençant la recherche à partir du nœud identifié par l'espace de nom contextuel.

Plus concrètement, lorsqu'on a une référence de type de la forme $M_1.M_2.\cdots.M_n.t$ qui annote un terme de `Typedlambda` dont l'espace de nom contextuel (toujours absolu) est $N_1.\cdots.N_p$, on va commencer par chercher le type t dans l'arborescence des types déclarés par l'unité de compilation (table `camil_typedecls`) au niveau du sous-module $N_1.\cdots.N_p.M_1.M_2.\cdots.M_n$. En cas d'échec, on cherche au niveau de $N_1.\cdots.N_{p-1}.M_1.M_2.\cdots.M_n$, et ainsi de suite jusqu'à trouver la référence absolue du type t, qui sera renvoyée par la transformation Θ.

Chapitre 3

Représentations des données et émission de code

Ce chapitre s'intéresse au *back-end* du compilateur OCamIL : le choix des représentations et la production de code.

Sommaire

3.1 Choix de représentations

Nous présentons les choix de représentations des valeurs qui ont été retenus dans l'implantation de OCamIL, dans les cadres respectifs de la reconstruction et de la propagation de types.

3.1.1 Représentation élémentaire des valeurs Caml (reconstruction de types)

Lorsque OCamIL fonctionne en mode reconstruction de types, les représentations possibles sont relativement élémentaires mais servent de base aux représentation plus sophistiquées autorisées par une propagation complète des informations de typage.

3.1.1.1 Types de base

Généralité : utilisation des types valeurs du CTS. Pour quels types Caml peut-on utiliser un type valeur ? Les types valeur sont munis d'une sémantique de copie (passage par valeur), il faut donc que les types Caml qu'ils incarnent présentent la même sémantique. Cela s'applique aux types Caml non mutables et dont la comparaison physique repose sur la valeur. C'est typiquement le cas des types de base (non structurés) qui, à l'exception de `string`[1], sont non mutables : `int`, `float`, `char`, `bool`, `int32`, `int64`, `native int` et `unit`. Ce qui compte ici n'est pas tant leur représentation dans l'environnement d'exécution Caml comme entiers ou blocs, mais leur sémantique.

L'intérêt est que bon nombre de types valeur sont directement exploitables par le jeu d'instructions de la machine virtuelle : par exemple les instructions arithmétiques portent sur les types numériques du CTS qui sont tous des types valeur. Il faut toutefois garder à l'esprit que les types valeur ne peuvent pas être utilisés dans n'importe quel contexte et qu'il faudra avoir régulièrement recours à des opérations de (dés)encapsulation.

Représentation des types de base. Le CTS dispose d'une palette de types suffisamment complète pour prendre en charge tous les types de base de Caml. Le problème vient uniquement du manque d'information sur les types dans l'approche par reconstruction, qui ne permet pas toujours d'utiliser les représentations les plus adaptées.

Le tableau suivant donne les correspondances possibles entre types de base Caml et types CTS :

1. En choisissant de conserver le caractère mutable de `string`, ce type ne pourra être implanté qu'au moyen d'un type référence.

Caml	*int*	*native int*	*int32*	*int64*	*bool*
CTS	IntPtr int32/int64	IntPtr	int32	int64	bool int8 identique à *int*
Caml	*float*	*unit*	*char*	*string*	
CTS	float64 float32	object (null) void type entier (0) ...	char int8 int16 identique à *int*	string System.Text.StringBuilder char[] int8[], int16[],IntPtr[]	

Ce tableau général est à mettre en correspondance avec la reconstruction effective des types Caml dans le langage `typeinfo` :

Caml	*int*	*native int*	*int32*	*int64*	*bool*
`typeinfo`	`TIint`	`TInint`	`TIint32`	`TIint64`	`TIint`
Caml	*float*	*unit*	*char*	*string*	
`typeinfo`	`TIfloat`	`TIunit/TIvoid`	`TIint`	`TIstring`	

Toutes les possibilités d'implantations ne seront pas autorisées en pratique puisque l'information contenue dans `typeinfo` est approximative.

Entiers. En ce qui concerne le type `int`, il n'y a pas à nos yeux de réel intérêt à conserver sa taille inhabituelle, qui est la conséquence d'un choix de représentation dans l'environnement de Caml. Deux choix possibles s'offrent alors à nous : utiliser le type `IntPtr` ou bien directement le type `int32` (sur une architecture 32 bits, et `int64` sur une architecture 64 bits). La première option a le mérite d'une portabilité totale, alors que la deuxième conduit à un code-octet différent suivant l'architecture cible. Les tests de performances que nous avons réalisé sur l'ensemble des primitives entières donnent les mêmes résultats sur les deux représentations (ce qui est normal en raison de la compilation JIT systématique qui va produire le même code *in fine* sur une architecture donnée).

Cependant le type `IntPtr` est dans les faits assez peu utilisé, et en particulier le langage C# ne propose pas de type intégral correspondant à `IntPtr` : celui-ci n'est utilisable que par le biais de son type référence, qui n'est pas lié aux différentes primitives (arithmétiques ou autres) opérant sur ses valeurs dans le jeu d'instructions de CIL. Un argument d'interopérabilité (considérant que C# est le plus gros fournisseur de composants .NET) nous a alors conduit à privilégier la deuxième possibilité dans l'implantation actuelle du compilateur OCamIL. De plus le programmeur désirant avoir la garantie de la taille de ses types entiers sur toutes les plates-formes peut toujours utiliser les entiers « spéciaux » que sont `native int`, `int32` et `int64`.

Les entiers « spéciaux » de Caml quant à eux sont directement implantés par les types correspondants `IntPtr`, `int32` et `int64`.

Booléens. Les booléens peuvent être représentés au moyen de `bool`, de `int8` (qui est en interne au CTS le type supportant l'énumération définie par `bool`) ou encore de la même représentation que celle choisie pour les entiers Caml. Dans le cadre de la reconstruction des types, nous devons choisir cette dernière possibilité, car il n'est pas toujours possible de distinguer les booléens des entiers.

Flottants. Les flottants Caml sont 64 bits, ce qui dans de nombreux langages (C, Java. . .) correspond au type `double` (soit `float64` dans le CTS) et non pas `float` (`float32` dans le CTS).

Unit. Le type `unit` a cela de particulier que son unique valeur `()` n'est presque jamais exploitée comme une valeur. Si dans l'environnement d'exécution de Caml la valeur `()` est représentée par l'entier 0 (ce qui est conforme avec l'implantation des types variants), on a le choix avec le CTS d'utiliser une valeur prédéfinie d'un type référence (naturellement la valeur `null`, qui sied à n'importe quel type référence) ou encore pas de valeur du tout (`void`). Nous avons choisi de représenter `unit` au moyen de `null` et `void` en fonction du contexte, la représentation choisie étant déterminée par la valeur `TIunit` ou `TIvoid` qui parvient au back-end du compilateur.

En ce qui concerne la distinction `TIunit`/`TIvoid`, notre politique est la suivante : `TIvoid` est un type indiquant qu'il n'y aura pas de valeur empilée (selon le dispositif décrit à la section 1.3.2.1) lors de l'évaluation de l'expression correspondante, alors que `TIunit` indique la présence d'une valeur (représentée par `null` à l'exécution). Par exemple :

- Les arguments de type `unit` d'une fonction sont toujours décorés par `TIunit` afin de gérer simplement l'application partielle.
- Une expression `for` ou `while` est décorée par `TIvoid`.

Si une confrontation `TIunit`/`TIvoid` se produit, elle est résolue statiquement : de `TIunit` vers `TIvoid` on émet une instruction `pop` et de `TIvoid` vers `TIunit` une instruction `ldnull` (qui place la valeur `null` sur la pile).

Caractères et chaînes de caractères. Les caractères Caml sont 8 bits et utilisent un encodage iso-8859-1 (latin1) alors que les caractères du CTS utilisent 16 bits chacun à l'encodage UTF-16 d'unicode. D'autre part, tout en ayant seulement 8 bits significatifs, les caractères Caml sont représentés par des entiers machine (donc au minimum 32 bits) lorsqu'ils sont isolés, mais occupent exactement 8 bits au sein d'une chaîne de caractères (formées d'un bloc spécial regroupant les caractères de manière compacte).

Pour l'implantation OCamIL, doit-on décider de coller à la représentation Caml, qui suggère d'utiliser `int8` ou `int32/int64`, ou bien de se rapprocher de la représentation naturelle dans le CTS (`char`, ou `int16`) ? Dans le cadre de la reconstruction de types, les caractères sont parfois indistinguables des entiers Caml, ce qui nous force à choisir la représentation `int32/int64`.

Le même problème se pose pour les chaînes de caractères, faut-il utiliser le type `string` natif de la plate-forme .NET ou alors des tableaux comme `int8[]`, `int16[]`

ou `IntPtr[]` ? Cette fois, il n'y a pas de problème lié à la reconstruction du type `string`, qui a une représentation bien séparée des autres types Caml grâce à l'emploi d'un bloc muni d'un tag spécial. Les premières options de représentation nous paraissent maladroites et nous préférons un type naturel dans le CTS, comme `string`, `char[]` ou `StringBuilder`.

Il faut maintenant avoir à l'esprit que les chaînes de caractères sont mutables en Caml alors que le type `string` ne l'est pas. Si F# a choisi d'utiliser ce type par souci d'efficacité, OCamIL quant à lui préfère en standard préserver les caractéristiques des chaînes Caml. Cela ne laisse que deux options : la classe standard `StringBuilder` qui propose des chaînes mutables, ou bien directement les tableaux de caractères `char[]`. S'il est plus simple d'utiliser la classe `StringBuilder`, qui propose de nombreuses méthodes de manipulation de chaînes, les tests de performance de la section 6.1.2.2 sont à l'avantage de `char[]`, que nous choisissons finalement.

Les options retenues sont résumées sur le tableau suivant :

Caml	int	native int	int32	int64	bool	float	unit	char	string
CTS	int32/64	IntPtr	int32	int64	int32/64	float64	null void	int32/64	char[]

Remarque : des options en ligne de commande du compilateur OCamIL permettent de choisir d'autres représentations des chaînes de caractères, à savoir `StringBuilder` et `string`. Des tests comparatifs de performances sont donnés aux sections 6.1.2.2 et 6.2.2.2.

3.1.1.2 Types algébriques

Les valeurs structurées que sont les n-uplets, les tableaux, les variants et les enregistrements, posent le problème de la représentation de données agrégées, qui sont hétérogènes dans la plupart des cas. Objective Caml fait une utilisation intensive des blocs car ils peuvent contenir des données hétérogènes en évitant souvent la pénalité des encapsulations. La représentation équivalente dans le CTS est le tableau d'objets `object[]`. Moyennant l'encapsulation de certains types, elle jouit de la même universalité mais entraîne des coûts assez lourds.

Dans le cadre de la reconstruction de types, seule la valeur `TIblock` de `typeinfo` est utilisée. Nous représentons celle-ci au moyen de tableau d'objets `object[]`. Plus précisément, un bloc de tag t et contenant n valeurs est implanté au moyen d'un tableau de taille $n+1$, où le tag est stocké à la fin du tableau. Ce choix rend coûteux l'exploitation du tag (puisqu'il faut calculer la longueur du tableau avant de pouvoir y accéder) mais en contrepartie simplifie la compilation des différentes expressions qui manipulent les blocs puisqu'on n'a pas besoin d'introduire de décalage sur les primitives d'accès aux champs.

La représentation par tableaux d'objets est universelle et très rapide à mettre en œuvre. Elle est bien sûr très peu optimale puisqu'elle provoque de nombreuses opérations d'encapsulation et de désencapsulation des types valeur. En l'absence d'informations de types détaillées c'est cependant la seule représentation possible qui garantisse une génération de code exempte d'erreur, parce qu'elle est extrêmement proche de la représentation des blocs dans l'environnement d'exécution standard de Caml.

3.1.1.3 Classes et objets Objective Caml.

Il est tentant de reproduire une hiérarchie de classes Objective Caml au moyen de classes du CTS. Cependant, cette voie est très compliquée à mettre en œuvre pour les raisons suivantes :

- les deux modèles objet mis en jeu sont radicalement différents (voir à ce sujet la section 4.2.1.1),
- la représentation interne des objets dans le code intermédiaire Objective Caml repose sur des structures plus bas niveau ; les objets et les classes sont directement implantés au moyen de blocs de champs et de fonctions si bien qu'il est difficile pour un back-end de retrouver la logique de la programmation objet au simple examen des langages intermédiaires. Par exemple le mécanisme de liaison tardive n'est pas pris en charge par la machine virtuelle de Caml et se retrouve être implanté directement dans le code engendré par le compilateur.

Dans ce contexte il est difficile de profiter des classes CTS autrement que comme de simples structures et il est impossible d'exploiter la liaison tardive de la plateforme cible. Nous nous sommes contentés dans l'implantation de OCamIL de compiler directement le code produit de manière habituelle par le compilateur Caml, si bien qu'aucun arrangement particulier n'est nécessaire pour la prise en charge des objets dans la compilation par reconstruction de types : dans ce cadre le compilateur OCamIL n'a même aucun moyen de savoir qu'il compile des objets Caml.

3.1.2 Représentation fine des valeurs Caml (propagation de types)

3.1.2.1 Types de base

Reprenons le tableau des correspondances possibles entre types de base Caml et types CTS :

Caml	*int*	*native int*	*int32*	*int64*	*bool*
CTS	IntPtr int32/int64	IntPtr	int32	int64	bool int8 identique à *int*

Caml	*float*	*unit*	*char*	*string*
CTS	float64 float32	object (null) void type entier (0) ...	char int8 int16 identique à *int*	string System.Text.StringBuilder char[] int8[], int16[],IntPtr[]

Dans le cadre de la propagation de types, le langage `typeinfo` n'introduit pas d'approximation sur les types de base, laissant leur chance à toutes les possibilités :

Caml	*int*	*native int*	*int32*	*int64*	*bool*
`typeinfo`	`TIint`	`TInint`	`TIint32`	`TIint64`	`TIbool`
Caml	*float*	*unit*	*char*	*string*	
`typeinfo`	`TIfloat`	`TIunit/TIvoid`	`TIchar`	`TIstring`	

Types inchangés. Comme on peut le voir sur le tableau précédent, le gain d'information obtenu par rapport à la version par reconstruction de types se situe au niveau des types `bool` et `char`. Pour tous les autres types de base, l'information est la même : ainsi nous conservons les choix précédents pour tous les types d'entiers Caml (`int`, `native int`, `int32` et `int64`), ainsi que les flottants, et `unit`.

Booléens. Il est possible de choisir une représentation plus adéquate des booléens, qui sont dans le cadre de la propagation de types, distinguables des entiers Caml. Nous optons pour le type `bool`, qui sans induire de dégradation de performances, a le mérite d'une lisibilité optimale.

Caractères et chaînes de caractères. Les limitations imposées par la reconstruction de types n'ont désormais plus cours : il nous faut reprendre la discussion sur la représentation des caractères. Rappelons que l'on peut soit utiliser une représentation semblable à celle de l'environnement d'exécution Caml (`int8` ou `int32/int64`) ou bien une représentation plus naturelle dans le CTS (`char`, ou `int16`).

Pour des questions d'efficacité d'implantation des primitives de manipulation de chaînes ainsi que d'interopérabilité (toutes les implantations .NET rencontrées reposent sur les caractères 16 bits du CTS), nous préférons la deuxième option, à base de `char`. Les caractères Caml sur 8 bits peuvent de toute manière tenir sur 16 bits, en acceptant de gaspiller un peu de mémoire.

En ce qui concerne les chaînes de caractères nous conservons le choix fait précédemment, à savoir `char[]`, qui est encore plus naturel lorsqu'on représente les caractères par le type `char`. Il est toujours possible de tester les représentations à base de `string` (plus efficaces mais non mutables) ou de `StringBuilder` à l'aide d'une option en ligne de commande du compilateur OCamIL. On pourra se référer à la section 6.1.2.2 pour comparer les performances respectives de ces alternatives.

Se pose le problème de savoir si on doit simuler le comportement des chaînes 8 bits de Caml (utilisant l'encodage iso-8859-1). La seule conséquence[2] de ce choix se situe au niveau de la sérialisation des chaînes de caractères : si on simule les chaînes iso-8859-1 dans OCamIL cela facilite la sérialisation de données entre programmes

2. Il y a aussi certaines optimisations dans le code de Caml qui reposent implicitement sur une implantation particulière des chaînes de caractères : c'est le cas des analyseurs lexicaux et syntaxiques de Caml qui utilisent des tampons 8 bits en interne et qu'il faut donc retoucher pour OCamIL.

compilés par les deux versions du compilateur. Il est possible de fournir un environnement d'exécution OCamIL alternatif qui tient compte de cette contrainte mais ce n'est pas l'environnement standard retenu pour OCamIL. L'interopérabilité avec les autres langages .NET et la plus grande ambition du standard unicode ont motivé ce choix : c'est désormais la norme dans l'industrie des langages et nous espérons qu'à terme l'implantation standard de Caml adoptera cette représentation.

Les options retenues sont résumées sur le tableau suivant :

Caml	int	native int	int32	int64	bool	float	unit	char	string
CTS	int32/64	IntPtr	int32	int64	bool	float64	null void	char	char[]

3.1.2.2 Types algébriques

Grâce à la propagation de types, nous avons à notre disposition toute l'étendue de la grammaire `typeinfo`, et il n'est plus obligatoire de représenter tous les blocs sous la forme de tableaux d'objets. Toutefois, malgré ses inconvénients, le type `object[]` nous servira de représentation refuge, quand il n'existera pas d'autre solution.

De nombreuses possibilités s'offrent à nous, mais nous devons tenir compte des quelques principes suivants : les représentations choisies devront être efficaces (tout en restant dans l'optique de produire du code géré) mais aussi suffisamment standard pour être exploitables dans le cadre de l'interopération; de plus une bonne lisibilité des structures retenues est utile pour utilisation en débogage (inspection directe des valeurs par exemple). D'autre part il faut garder à l'esprit que des types Caml sans définition ne pourront être représentés par des types CTS utilisateurs (nous avons discuté des contraintes de la compilation séparée dans la section 2.1.2.1).

Types enregistrement. Les enregistrements sont entièrement déterminés par leur définition. Nous choisissons de les implanter au moyen de classes dont les champs reproduisent les champs des enregistrements (à la fois par leur nom et leur type). Dans le cas d'un enregistrement paramétrique, chaque instance de variable de type est remplacée par le type multi-usage `object`. Le module comportant la définition de type provoque la génération d'une définition de type CTS (voir la section 3.2.1), qui hérite de la classe `CamIL.Record` (définie dans le runtime OCamIL) servant de classe mère à tous les enregistrements. Cette classe définit, outre un constructeur par défaut, quatre méthodes virtuelles d'instance :

- `int32 compare(CamIL.Record)` et `bool equals(CamIL.Record)` sont utilisées par les fonctions de comparaison polymorphes,
- `CamIL.Record duplicate()` est utilisée à l'exécution pour la copie d'enregistrements (afin d'implanter le mot-clé `with` par exemple),
- `int32 hashcode()` est appelée par les routines du module `Hashtbl`.

L'implantation de ces méthodes dans chaque type enregistrement reproduit au mieux le comportement de Caml (en s'inspirant du code C qui supporte les fonctions correspondantes sur les blocs dans l'environnement d'exécution standard). Par exemple :

Type Caml	Type CTS

`type people = {name:string; age:int}`	`class people extends CamIL.Record {`
	`char[] name;`
	`int32 age;`
	implantations méthodes virtuelles
	`}`

`type 'a label = {lbl:string; v:'a}`	`class label extends CamIL.Record {`
	`char[] lbl;`
	`object v;`
	implantations méthodes virtuelles
	`}`

Le choix d'un type référence est en ligne avec la sémantique de Caml, dont le runtime représente les enregistrements par des blocs soumis au récupérateur automatique de mémoire. Cela permet également l'implantation aisée des enregistrements mutables et une représentation homogène des structures récursives.

Types variant. Il faut ici distinguer les types variant, complètement déterminés lors de leur définition, des types variant polymorphes. Ces derniers ne peuvent faire l'objet d'une implantation au moyen d'un type utilisateur et c'est pourquoi nous utilisons pour eux la représentation refuge `object[]` des tableaux d'objets.

Intéressons-nous aux variants simples : nous les implantons au moyens de classes héritant de `CamIL.Variant` définie dans le runtime OCamIL. Nous voyons les variants comme une famille prédéterminée d'enregistrements (discriminés entre eux par un champ de tag) et dont les noms de champs ne sont pas définissables par l'utilisateur (et qui seront nommés conventionnellement `x0`, `x1` et ainsi de suite). La classe `CamIL.Variant` introduit :

- Un champ `tag` de type `int32`.
- Des méthodes virtuelles d'instance `compare`, `equals` et `hashcode` analogues à celles des enregistrements. En particulier les deux méthodes de comparaison supposent de travailler sur deux instances *de même tag* du type variant.
- Des méthodes d'instance `vcompare` et `vequals` qui effectuent d'abord une comparaison générique basée sur le tag des deux instances à comparer, puis en cas d'égalité, délèguent aux méthodes de comparaison précédentes.
- Un constructeur prenant en argument le tag de l'instance à construire (passée comme un entier `int32`).

Si on prend l'exemple du type :

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

alors le compilateur génère les classes comme sur la figure 3.1, possédant les champs suivants :

Classe	**Tag**	**Champs**
`partialtree`	*abstraite : aucun champ; pas de tag ni de constructeur*	
`DeadEnd_OF_partialtree`	0	*aucun*
`Leaf_OF_partialtree`	1	`x0:object`
`Node_OF_partialtree`	2	`x0:partialtree` et `x1:partialtree`

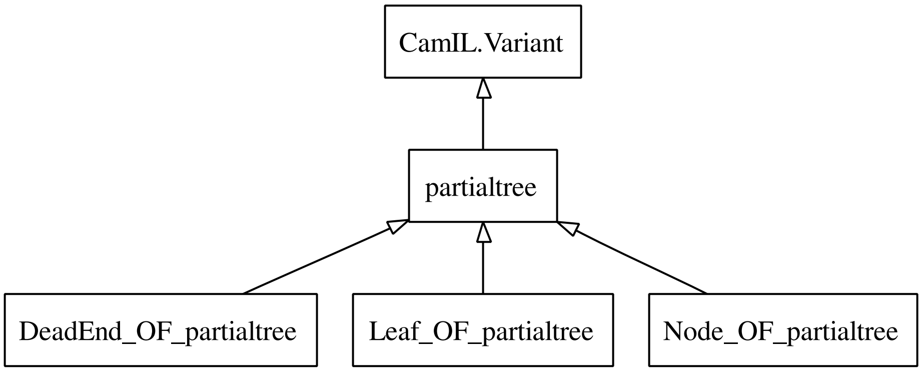


FIG. 3.1 – *Un exemple du type variant* `'a partialtree`

L'utilisation d'un nom de la forme *constructeur*`_OF_`*type-variant* permet de reporter les problèmes de conflits de noms potentiels au niveau des types déclarés par Caml eux-mêmes, qui sont uniques dans un module (voir la section 3.1.4).

Remarques:

– Il n'est pas obligatoire de définir un type correspondant au type variant au dessus des types mis en place pour les différentes alternatives du variant (comme la classe `partialtree` ci-dessus), car il n'a rien à introduire de plus que `CamIL.Variant`. Cependant l'intérêt réside dans la lisibilité du code pour le débogage, puisqu'un emplacement mémoire tel qu'un argument de fonction pourrait recevoir ce type plus précis que `CamIL.Variant`. L'apparente indirection supplémentaire lors de la résolution de la liaison tardive vers les méthodes de `CamIL.Variant` peut être levée en ayant recours à des appels non virtuels.

– Nous utilisons toujours la même optimisation que dans le cadre de la reconstruction de types, à savoir de représenter les variants formés uniquement de constructeurs constants au moyen d'entiers. Une nouveauté ici est de pouvoir définir un type énumération basé sur les entiers, et dont les alternatives reçoivent les noms donnés au niveau de la définition Caml. L'intérêt ne réside ici encore que dans la lisibilité des représentations, et il faut évaluer l'impact sur les performances de ce choix.

Les variants introduits par le noyau du langage Caml (tels que les types `option` et `list`) sont compilés suivant le même processus que les variants déclarés par du code utilisateur. Le langage de types `typeinfo` les distingue afin d'autoriser des représentations alternatives et plus optimisées pour ces types très utilisés. Il est par exemple possible d'utiliser pour les listes des implantations natives de l'environnement .NET même si ce n'est pas le cas actuellement dans le prototype du compilateur OCamIL.

Tableaux. Remarquons que les représentations intermédiaires de OCamIL propagent une information complète sur le type des tableaux puisque la description `TIarray` de `typeinfo` prend en argument le type des éléments des tableaux. Peut-on alors faire mieux que la représentation passe-partout `object[]` et utiliser des représentations plus typées (comme `int[]`, `float[]` etc...) pour des tableaux moins génériques ? On serait alors plus proche de l'implantation standard de Caml qui a des représentations adaptées pour les tableaux de flottants, mais aussi naturellement pour les tableaux d'entiers par le biais de la représentation avec tag. On pourrait obtenir un gain d'efficacité important en éliminant des opérations d'encapsulation et de désencapsulation autour des accès aux éléments des tableaux, mais puisque OCamIL ne monomorphise pas les fonctions polymorphes, il faut utiliser une interface générique masquant les implantations spécialisées des tableaux. L'emploi de `System.Array` (sur-type de tous les tableaux possibles) n'est pas idéal car l'accès aux éléments d'une valeur de ce type ne se fait plus directement au moyen d'instructions CIL mais via l'appel de méthodes `GetValue` et `SetValue`, qui s'avèrent plus coûteuses. D'autre part l'utilisation d'une interface particulière à OCamIL associée à des implantations spécialisées conduirait à une représentation non-standard des tableaux, ce qui limite l'interopérabilité.

Compte tenu des inconvénients précédents, la version actuelle du compilateur OCamIL utilise la représentation générique sous forme de tableaux d'objets `object[]` (mais la case additionnelle réservée pour le tag des blocs génériques est supprimée, ce qui permet d'avoir une structure standard de la bonne longueur, directement adaptée à l'interopérabilité). L'amélioration envisagée dans le futur pour la représentation des tableaux (et qui est aussi valable pour les n-uplets) passe par l'exploitation des *Generics*.

Implantation du module `Obj`. Le module `Obj`, introduit dans la bibliothèque standard mais non documenté, pose un nouveau défi à la représentation des valeurs. Ce module permet de jouer sur les représentations mémoire des valeurs Caml en contournant le système de types. Cela permet des optimisations dangereuses mais parfois très utiles : la bibliothèque standard de Caml y a elle-même recours à de nombreuses reprises.

Il est possible d'émuler parfaitement le module `Obj` uniquement lorsque la représentation des blocs est uniforme, ce qui cesse d'être le cas quand on cherche à optimiser les représentations dans le CTS. Certaines opérations restent possibles (comme celles qui inspectent les blocs) mais toute opération visant à muter une valeur en mémoire et ayant pour effet de modifier le type de sa représentation est hors de portée.

Le module est maintenu dans la bibliothèque de OCamIL car certaines opérations continuent à fonctionner : inspection de bloc (supposant d'ailleurs que le code réalise dynamiquement un test de type sur la valeur inspectée afin de brancher sur l'opération adéquate dans le monde .NET) et transtypage « sauvage » via `Obj.magic` dans certains cas simples (où on a recours à `object` comme type transitoire). Cependant l'utilisation de ce module reste fortement déconseillée et peut conduire le compilateur à générer du code incorrect.

3.1.2.3 Classes et objets Objective Caml.

Dans le cadre de la propagation de types, la compilation des classes et objets n'est guère différente de la compilation avec reconstruction de types. Les difficultés à faire rentrer le modèle objet original de Objective Caml dans celui plus classique du CTS sont toujours présentes. Au niveau de l'implantation, nous devons veiller à propager correctement les types classes et objets sous forme de types structurés (enregistrements ou tableaux d'objet).

3.1.3 Application et structures de contrôle

Décider de l'implantation des structures de contrôle est indissociable de la représentation des valeurs puisque certains types de Caml, tels que les types flèches et le type `exception`, sont au cœur même des mécanismes de contrôle. De plus les choix de représentation des différentes structures de contrôle a un impact décisif sur les performances du code engendré.

3.1.3.1 Fermetures et application

Nous détaillons ici la représentation des fermetures, des fermetures partagées, ainsi que la réalisation de l'application et de l'application partielle.

Fermetures : représentation, création et application. L'environnement d'exécution de Caml utilise des blocs pour représenter les fermetures, alors que pour OCamIL nous utilisons des classes du CTS.

Partons de l'exemple suivant :

```
let func d =
  let rec frec x y =
    if x <= 0 then y else x * (frec (x - d) (y + 1))
  in frec
```

Les classes utilisées par OCamIL suivent le diagramme d'héritage de la figure 3.2.

Pour chaque fermeture syntaxique dans le code source d'un programme Caml (ce qui exclut donc les fermetures construites dynamiquement du fait de l'application partielle, voir la section 2.1.1.2), comme ci-dessus `func` et `frec`, une classe est générée; celle-ci héritant directement de la classe `CamIL.Closure`, ou bien de la classe `CamIL.PappClosure` quand l'application partielle doit être prise en charge (c'est-à-dire quand la fonction n'est pas totalement décurryfiée et a au moins deux arguments).

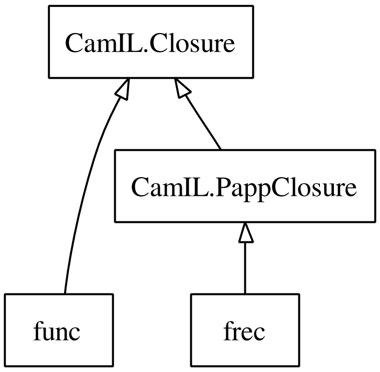


FIG. 3.2 – *Hiérarchie des fermetures*

La classe de base `CamIL.Closure` déclare essentiellement une méthode d'instance virtuelle `object apply(object)` qui est en charge de l'application générique, c'est-à-dire de l'application de la fonction sous forme décurryfiée à exactement un argument. On peut voir `CamIL.Closure` comme une interface commune à toutes les fermetures : celle-ci est nécessaire pour typer les applications génériques qui n'ont pas connaissance statiquement de la fonction mise en jeu (par exemple dans le code d'une fonctionnelle comme `List.map` qui ignore l'identité de son premier argument, une fonction, mais qui doit savoir l'appliquer à son deuxième argument, une liste).

La classe `CamIL.PappClosure` déclare un champ `nargs:int32` qui tient le compte du nombre des arguments déjà partiellement appliqués, et un champ `args:object[]` qui stocke ces arguments sous une forme générique. La classe fournit également des fonctions élémentaires de manipulation de cet environnement, telle la méthode statique `gen_apply` dont nous illustrons le fonctionnement plus loin. Il faut bien noter que l'environnement d'une fermeture (qui n'est pas formé de ces arguments pré-appliqués mais des variables libres dans la déclaration de la fonction), est stocké séparément, comme détaillé ci-après.

Chaque fermeture particulière étend sa classe de base en :

– Définissant une méthode statique `exec` contenant le code de la fonction décurryfiée : le prototype de cette fonction est typé de la manière la plus proche possible de la fonction Caml à compiler; par exemple pour la fonction `String.get` de la bibliothèque standard ce sera `char exec(char[], int32)` (en supposant que l'on représente les entiers par `int32` et les caractères par `char`). C'est l'analogue du pointeur de code vers la fonction pour l'environnement Caml traditionnel, voir la section 2.1.1.2.
– Implantant la méthode `apply` par du code capable d'appeler la méthode `exec` en réalisant les conversions de types qui s'imposent sur les arguments et le résultat.
– Déclarant éventuellement des champs de classe destinés à contenir l'environnement de la fermeture (les variables libres dans la déclaration de la fonction) : ces champs sont typés le plus finement possible. On introduit aussi un constructeur prenant en arguments les valeurs à placer dans cet environnement.

Revenons à l'exemple de `func` et `frec`. Les méthodes `exec` ont respectivement pour prototypes :
- `CamIL.Closure func::exec(int32)`
- `int32 frec::exec(int32, int32, class frec)`

On peut remarquer que :
- Comme `frec` a besoin de son environnement, un argument additionnel permet le passage de celui-ci lors de l'exécution. Ici ce n'est pas lié au caractère récursif de la fonction car elle s'auto-appelle au moyen d'une application directe.
- Le type Caml de `func`, `int->int->int`, est en réalité décomposé comme `int->(int->int)` et par la suite traduit sous la forme `int32->CamIL.Closure`. En effet appliquer deux arguments à `func` est un cas de sur-application : il faut d'abord appliquer le premier argument, récupérer une fermeture (ici une instance de `frec`) et ensuite lui appliquer le deuxième argument.

La méthode `func::apply` est très simple : elle désencapsule son argument pour obtenir un entier et ensuite appelle la méthode `func::exec`.

La méthode `frec::apply` teste d'abord la valeur du champ `nargs`. Si celui-ci est nul, alors elle crée une copie de la fermeture et de son environnement, et appelle `PappClosure PappClosure::gen_apply(PappClosure,PappClosure,object)`, une méthode statique qui configure la copie en recopiant le tableau des arguments pré-appliqués `args` (trivial dans cet exemple), en y ajoutant le nouvel argument passé à `apply` et en incrémentant le champ `nargs`. Dans l'autre cas de figure, le champ `nargs` vaut 1, et dans ce cas tous les arguments effectifs sont extraits de `args`, désencapsulés, et passés à `frec::exec` (sans oublier l'argument additionnel contenant la fermeture elle-même).

En ce qui concerne les applications de fermetures, celles-ci peuvent être de deux types : directe ou générique.

Dans le cas d'une application directe, le contexte est suffisant à la compilation pour générer un appel direct au code de la fonction (cela correspond à l'instruction `Udirect_apply` de `Clambda`) : nous nous contentons dans ce cas d'appeler la méthode `exec` de la fermeture en lui fournissant tous ses arguments, y compris l'argument additionnel référençant la fermeture lorsque c'est nécessaire.

Dans le cas d'une application générique (instruction `Ugeneric_apply`), les arguments disponibles, qui peuvent être en nombre exact, en sous-nombre (cas de l'application partielle) ou en surnombre (cas de la sur-application), seront tous convertis en types objet et placés sur la pile au dessus d'une instance de fermeture au cours d'une boucle appelant plusieurs fois une méthode d'instance `apply`. Dans le cas de l'application partielle et de la sur-application, cela nécessite de transtyper chaque résultat intermédiaire de type `object` vers le type `CamIL.Closure`, instance sur laquelle est appelée la méthode `apply` suivante. La valeur de retour est ultimement transtypée depuis `object` vers le type attendu.

Remarques :

- La méthode `exec` est statique, ce qui est tout à fait cohérent avec la représentation traditionnelle des fermetures Caml : lorsque l'environnement de la fermeture est nécessaire dans le code de la fonction, le compilateur Caml introduit un argument supplémentaire qui est une référence vers le bloc fermeture. Nous avons conservé cette approche au lieu de nous baser sur une méthode d'instance pouvant directement fouiller dans l'environnement, car cela permet un branchement plus facile sur le code `Clambda` fourni par les premières passes de la compilation, et aussi parce que l'utilisation d'une méthode statique évite le mécanisme de liaison tardive sur l'appel de `exec`, ce qui nous fait gagner en efficacité sur le domaine crucial de l'application des fonctions.
- La sémantique opérationnelle de OCamIL concernant l'application diffère légèrement de celle de Objective Caml. Alors que l'implantation de ce dernier évalue les arguments de la droite vers la gauche, celle de OCamIL évalue les arguments de gauche à droite, ce qui est plus commode au regard de la pile de la machine .NET. À ce sujet, le langage Objective Caml laisse explicitement l'ordre d'évaluation non spécifié, laissant le choix aux implantations. Cette différence a de l'importance dans le cas d'arguments qui comportent des effets de bord. Pour remédier à tout problème éventuel, le compilateur OCamIL est muni d'une option en ligne de commande permettant de forcer l'évaluation des arguments de fonctions à suivre l'ordre de droite à gauche. On pourra trouver à la section 6.2.2.2 le détail de l'impact de cette option sur les performances.

Le cas des fermetures partagées. Prenons l'exemple de deux fonctions mutuellement récursives `frec1` et `frec2` ayant des variables libres dans leurs déclarations. Caml utilise une fermeture partagée possédant des références vers le code des deux fonctions et un environnement commun. Afin de réutiliser les interfaces précédentes (`CamIL.Closure` et `CamIL.PappClosure`), ce qui est nécessaire pour prendre en charge l'application générique, nous proposons une représentation comme sur la figure 3.3 (les flèches pleines représentent des références et les flèches vides la relation d'héritage).

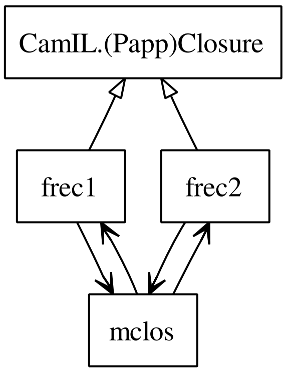


FIG. 3.3 – *Un exemple de fermeture partagée*

Nous introduisons une classe supplémentaire `mclos` destinée à recevoir l'environnement partagé, implanté comme des champs de classes typés le plus précisément possible. Des champs sont utilisés pour référencer les instances de `frec1` et `frec2`, et réciproquement l'environnement de celles-ci est réduit à une référence sur la fermeture partagée. La traduction des instructions `Pfield` et `Uclosure` sur la fermeture partagée nécessite des indirections via ces références.

Dans le cas où les fonctions mutuellement récursives n'ont pas besoin d'environnement, les champs servant à référencer la fermeture partagée `mclos` ne sont pas engendrés dans les classes de fonctions.

Implantations alternatives des fermetures et optimisations. Les fermetures étant essentiellement des structures de données regroupant des pointeurs de fonctions et des champs d'un environnement, on peut envisager des implantations différentes selon la technique utilisée pour la liaison et l'appel de la fonction d'une part (principalement au moyen d'une interface ou d'un délégué) et la représentation des champs d'autre part (soit fortement typée soit générique). Si on se place dans un cadre où une instance de fermeture est représentée par un objet, on peut distinguer les deux approches suivantes :

- Si la classe de l'objet est spécifique à cette fermeture, la fonction est naturellement implantée par une méthode de la classe ayant le prototype le plus fidèle possible et l'environnement peut être fortement typé.
- Si on utilise une même classe pour implanter des fermetures diverses, celle-ci doit avoir accès à une liste de méthodes aux prototypes différents et s'occuper elle-même de la liaison d'un appel à la bonne méthode. L'environnement est quant à lui générique (tableau d'objets).

C'est la première solution qui est utilisée par OCamIL : on associe à chaque fermeture syntaxique une classe dédiée permettant le typage le plus précis possible. Même en faisant ce choix, l'appel dynamique de fonctions dans le cas où seul le type de la fonction (et pas l'identité exacte de la fermeture) est connu au niveau de l'appelant peut être abordé de différentes manières :

- Passer par une interface générique (à la manière de la méthode `apply` de la classe `CamIL.Closure`). On peut aussi définir une famille d'interfaces qui se chargent des différentes arités.
- Passer par un type délégué générique (on peut également définir une famille de délégués génériques qui se chargent des différentes arités). Cependant les mesures de performances sont à l'avantage des interfaces.
- Utiliser une interface qui déclare une méthode du bon type : les classes fermetures doivent déclarer explicitement qu'elles implantent cette interface. Il se pose le problème de la détermination statique des interfaces utiles, surtout dans un contexte de compilation séparée : combien d'interfaces doit-on déclarer et où doit-on les déclarer ?
- Utiliser un type délégué correspondant au type de la fonction. Les délégués sont plus souples que les interfaces car on peut les instancier sur des méthodes arbitraires (pourvu qu'elles aient le bon type) sans pour autant que la

classe déclarant la méthode à appeler ait besoin de connaître le type délégué à l'avance (voir la section 1.1.3). La question est de savoir si on est forcé de définir exactement un type délégué par signature de fonction possible. C'est malheureusement le cas : l'appelant ne peut utiliser un type délégué local car pour l'instancier il faut pouvoir référencer précisément la méthode qui implante le code de la fermeture, et donc connaître la classe exacte de la fermeture (ce qui n'est pas possible lors d'une application de fonction générique). Il ne reste qu'à instancier le délégué au niveau de la construction de la fermeture mais alors tous les appelants doivent utiliser ce même type délégué et on retombe sur le problème de détermination statique mentionné ci-dessus pour les interfaces, et fait perdre l'avantage des délégués.

La deuxième approche (utilisation d'une seule classe pour représenter les fermetures) est suivie par l'adaptation .NET du compilateur Bigloo pour le langage Scheme. L'article [11] contient une analyse comparée de différentes techniques d'implantation des fermetures dans Bigloo. Au départ porté vers le C et Java, Bigloo a dû imaginer des alternatives à la représentation des fermetures par autant de classes déclarées : en effet dans Java chaque classe correspond à un fichier séparé, ce qui pose alors d'importants problèmes de performances.

Bigloo représente autant que possible les fonctions par des méthodes statiques : c'est possible pour les fonctions sans environnement et qui ne sont pas utilisées comme des citoyens de première classe (c'est-à-dire qui ne se trouvent pas comme argument ou retour d'une fonction et ne sont pas stockées dans des structures de données). Pour traiter le cas général, chaque module définit une classe unique pour toutes les fermetures qu'il contient. Les fonctions sont implantées par des méthodes de cette classe et font référence à un environnement non typé, implanté comme un tableau d'objets membre de la classe (remarquons que le fait que Scheme est un langage typé dynamiquement rend plus naturelle cette représentation des environnements, et simplifie la compilation sur .NET). Les instances de cette classe renseignent un champ entier qui identifie la fonction sous-jacente à la fermeture. Les opérations d'application générique sont typées au moyen du type multi-usage `object` sur l'ensemble des arités possibles pour les fonctions du module. Ces opérations contiennent un code de dispatch qui se sert de l'identifiant entier contenu dans chaque instance de fermeture pour appeler la méthode contenant le code de la fermeture (le code de dispatch est déterminé statiquement à la compilation de chaque module pour lequel on connaît l'ensemble des fonctions).

Une alternative consiste à remplacer l'identifiant entier par une référence à un délégué : à la construction de la fermeture on crée une instance de la classe fermeture générique dans laquelle le champ de type délégué encapsule l'appel à la méthode qui implante la fonction. Ainsi le code de dispatch basé sur les entiers est remplacé par un appel direct au délégué. Il faut prévoir autant de délégués que d'arités possibles au sein du module.

Les tests reproduits dans [11] indiquent que la solution la plus efficace sur les différentes machines .NET est le code de dispatch basé sur des appels indexés sur les entiers, suivie de l'implantation par délégués, et en dernière position l'implantation à base de classes dédiées à chaque fermeture (celle-ci étant coûteuse à cause du temps pris par le chargeur de classes et ses opérations de vérification). Cependant la situation est différente entre Scheme et un langage typé statiquement comme Caml, où l'on a intérêt à typer fortement les valeurs de l'environnement (pour des questions d'efficacité et de lisibilité), et donc à définir un type adapté à chaque situation.

Dans un tout autre ordre d'idées, l'environnement .NET permet l'utilisation de pointeurs sur des méthodes, mais ceux-ci ne sont pas utilisables dans du code vérifiable, aussi l'avons-nous écartée (Bigloo a fait de même).

Certaines optimisations proposées dans [11] peuvent être mises en œuvre dans OCamIL. Par exemple implanter directement par des méthodes statiques les fonctions locales qui ne sont pas utilisées comme des citoyens de première classe. Bigloo va même jusqu'à les insérer en-ligne lorsque leur code est assez petit. En ce qui concerne les fonctions uniquement utilisées lors d'appels récursifs terminaux, Bigloo choisit de les coder directement comme des boucles. OCamIL optimise quant à lui les appels récursifs terminaux d'une méthode sur elle-même en rebranchant sur la première instruction de la méthode après avoir écrasé les arguments par leurs nouvelles valeurs (instruction `starg`). Dans le cas général, Bigloo comme OCamIL ont recours au préfixe `.tail` lors des appels récursifs terminaux afin de ne pas saturer la pile avec des cadres d'activation superflus. Il semblerait cependant d'après les tests de [11] que cela tendrait à diminuer la vitesse d'exécution des programmes. Une autre voie serait l'utilisation de l'instruction `jmp`.

Une voie intéressante est l'utilisation du toolkit ILX [82], une rustine développée pour le CLR afin de le munir d'une représentation native des fermetures. Nous avons été retenus par le manque de tests de performances et le caractère non-standard de cette extension (elle n'est plus maintenue). ILX n'est finalement qu'une sorte de sucre syntaxique puisque son implantation traduit ses nouvelles représentations vers des constructions standard de la plate-forme. Malgré tout, il serait très intéressant pour les langages fonctionnels que le travail sur ILX soit poursuivi de manière à munir l'environnement d'exécution .NET de constructions dédiées aux langages fonctionnels. En terme d'interopération, cela fournirait un cadre commun (et on l'espère, efficace) pour l'implantation des fermetures.

De son côté, le langage C# introduit des lambda-expressions dans sa dernière version 3.0. Il est possible de définir des fonctions dans des expressions qui capturent leurs variables libres dans une fermeture. Un mécanisme embryonnaire d'inférence de types a même été implanté. Notons qu'il n'y a pas d'application partielle (il faut créer à la main les fermetures résultant de l'application partielle) et qu'il faut déclarer au préalable les types fonctionnels utilisés au moyen de délégués.

L'implantation de C# 3.0 se base uniquement sur les possibilités de la plate-forme 2.0. Des classes sont engendrées pour chaque fermeture : une méthode implante la fonction et des champs capturent l'environnement. L'instanciation de fermetures consiste à créer un objet délégué en lui passant une référence sur la méthode de la fonction. Malgré l'utilisation des délégués c'est une approche assez semblable à celle de OCamIL. En revanche nos tests n'ont pas mis en évidence d'optimisation des appels récursifs terminaux.

3.1.3.2 Processus légers (threads)

Les processus légers (*threads*) de Objective Caml sont implantés directement au moyen des threads .NET : ainsi le type abstrait `Thread.t` de la bibliothèque `threads` de Caml encapsule la classe `System.Threading.Thread` dans l'implantation OCamIL.

Le modèle de threads de la plate-forme est totalement similaire à celui de Caml, si bien qu'il a été possible d'implanter la majorité des fonctions des modules `Thread`, `Condition`, `Event` et `Mutex` de la bibliothèque `threads` de Caml directement à partir de méthodes de la BCL. Cette adéquation simple sera d'une grande utilité dans le cadre de l'interopérabilité.

Afin de prendre en charge les threads, la classe `CamIL.Closure` déclare une méthode d'instance supplémentaire `void threading_delegate()` qui exécute la méthode `apply` de la fermeture sur un argument `null` (cette méthode n'a de sens véritable que pour les fonctions Caml de type `unit->unit` : il est nécessaire de normaliser les fonctions d'appel des threads).

En pratique, la fonction `Thread.create: ('a -> 'b) -> 'a -> Thread.t` qui démarre un nouveau thread (exécutant l'application de fonction spécifiée en arguments) est mise en œuvre de la manière suivante dans la bibliothèque OCamIL (appel de `Thread.create f x`) :

1. on crée une nouvelle fermeture `f' = fun () -> ignore (f x)` dont le type est standardisé en `unit -> unit`,
2. on récupère un délégué sur `CamIL.Closure::threading_delegate` de la fermeture `f'` sur lequel on construit un objet `System.Threading.ThreadStart`[3],
3. on appelle le constructeur de thread `System.Threading.Thread::.ctor` sur cet objet et on donne un nom au thread provenant d'un générateur d'identificateurs entiers uniques (les *id* de threads en Caml sont des entiers),
4. enfin on démarre le thread en appelant sa méthode `Start` et on retourne une référence sur ce thread.

Pour le reste la classe `Threading.Thread` fournit des méthodes qui correspondent presque parfaitement aux fonctions déclarées dans le module Caml `Thread` :

3. Depuis la version 2.0 de .NET il est possible d'utiliser un objet `ParameterizedThreadStart` qui accepte un délégué de type `void delegate(object)` qui évite la manipulation de fermetures décrite ici.

Fonction Caml	**Méthode BCL**
`self: unit -> Thread.t`	`Thread get_CurrentThread()` (statique)
`id: Thread.t -> int`	`string get_Name()`
`join: Thread.t -> unit`	`void Join(Thread)` (statique)
`kill: Thread.t -> unit`	`void Abort()`
`exit: unit -> unit`	`void Abort()` (appelée sur `self`)
`delay: float -> unit`	`void Sleep(System.TimeSpan)` (appelée sur `self`)
`sleep: unit -> unit`	`void Interrupt()` (appelée sur `self`)
`wakeup: Thread.t -> unit`	`void Resume()`

Sans détailler davantage, signalons que les mutex Caml s'implantent avec la même facilité grâce aux services proposés par la classe `System.Threading.Mutex` de la BCL. Les modules `Condition` et `Event` sont bâtis au dessus de `Thread` et `Mutex` au moyen de code Caml uniquement et sont donc pris en charge sans difficulté dans l'implantation OCamIL.

3.1.3.3 Exceptions

On pourra consulter l'article [40] pour une analyse des exceptions .NET.

Implantation des valeurs d'exception. Les exceptions (ou ruptures de contrôle) de Objective Caml sont directement implantées au moyen du mécanisme d'exceptions de la plate-forme cible. Pour cela on utilise une classe `CamIL.Exception` qui hérite de `System.Exception`, la classe racine de toutes les exceptions du CTS.

Ce choix présente l'avantage de la simplicité et autorise une bonne intégration des composants Caml dans un contexte d'interopérabilité. Chaque classe héritant de `CamIL.Exception` contient une représentation du bloc contenant l'identité de l'exception et ses paramètres. La contrainte est de pouvoir discriminer les différentes instances d'exceptions différentes (pour implanter le récupérateur `with...`) et ensuite d'accéder aux champs des arguments.

Si une implantation analogue à celle des variants semble la plus prometteuse en termes de performances, nous avons choisi de nous tenir à la représentation élémentaire sous forme de tableau d'objets. La raison est pragmatique et tient à la grande médiocrité des performances des exceptions dans l'environnement .NET, comme nous le verrons en détail dans la section 6.1.3.2. Toute optimisation sur les valeurs d'exceptions aura de toutes façons un impact dérisoire comparé aux pertes dues au mécanisme d'exception lui-même. Ici nous avons simplement un champ `v1: object[]` dans la classe `CamIL.Exception` qui reproduit exactement le bloc Caml de l'exception (le premier champ référence le nom du constructeur d'exception sous la forme d'une chaîne de caractères).

Structures de contrôle des exceptions. Afin d'implanter la construction `try ...with` nous utilisons les bloc `try{...}` et `catch{...}` de CIL. Ceux-ci imposent un certain nombre de restrictions :

– la pile doit être vide à l'entrée d'un bloc `try`,

– la pile doit être vide à la sortie d'un bloc `try` ou d'un bloc `catch`,
– on ne peut sortir d'un bloc `try` ou `catch` qu'au moyen d'une instruction `leave` ou du lancement d'une exception (instructions `throw` et `rethrow`).

Compte tenu de ces contraintes, le code engendré pour une expression `Clambda Utrywith(t1,e,t2)` a l'allure suivante :

```
CODE SAUVANT L'ÉTAT DE LA PILE DANS DES VARIABLES LOCALES
try {
   CODE COMPILÉ POUR t1
   stloc VAL //sauver le résultat de t1 dans une variable locale VAL
   leave EXIT
}
catch CamIL.Exception {
   //la valeur d'exception est sur la pile
   ldfld object[] CamIL.Exception::v1
   stloc VAR_E //utilisé par t2 (où la variable e est libre)
   CODE COMPILÉ POUR t2
   stloc VAL //sauver le résultat de t2 dans une variable locale VAL
   leave EXIT
}
EXIT:
CODE RÉCUPÉRANT L'ÉTAT DES VARIABLES LOCALES
ldloc VAL //récupérer le résultat de l'évaluation du try..with
```

Le code de `t2` a la responsabilité de relancer les exceptions non traitées; il le fait au moyen de l'instruction `rethrow`. Nous verrons à la section 4.2.2.1 comment une légère modification de ce schéma permet de capturer des exceptions externes à Caml.

Implantations alternatives. L'article précisant le fonctionnement interne du compilateur MoscowML [51] contient une discussion sur la représentation des exceptions. Les développeurs ont tenté de laisser tomber les exceptions .NET au profit d'une gestion ad-hoc. Celle-ci consiste à pouvoir retourner une valeur d'exception comme une alternative aux valeurs de résultats de fonctions. Le code appelant une fonction serait alors responsable d'analyser la valeur de retour et de mettre en œuvre un mécanisme de récupération ou de transmission de l'exception. Cette solution a semble-t-il permis de gagner en efficacité pour des programmes faisant une utilisation intensive des exceptions, cependant elle n'a pas été retenue pour OCamIL. En effet d'une part ce schéma dégrade les performances de code n'utilisant pas les exceptions comme moyen de contrôle, et de plus cela entraîne de trop grosses modifications du code engendré : celui-ci devient fortement non-standard (à la fois par l'incompatibilité avec les exceptions usuelles et par l'alourdissement des prototypes de fonctions) ce qui est nuisible pour interopérer.

On peut noter que Bigloo compile `call/cc` au moyen des exceptions de la plateforme. Il ne propose pas un support complet des continuations sur le back-end .NET (ni Java pour la même raison) en l'absence d'instructions autorisant l'accès complet de la pile en écriture. Les continuations autorisées ont un pouvoir expressif identique aux exceptions (elles ne peuvent pas être utilisées pour suspendre et reprendre des calculs).

3.1.4 Modules et foncteurs

D'un point de vue physique, un programme Objective Caml est structuré en modules d'implantations et fichiers d'interfaces. D'un point de vue logique, il est structuré en modules et sous-modules, les modules pouvant être des abstractions sur d'autres modules (foncteurs).

Le langage de modules de Caml a pu être pris en charge dans OCamIL au contraire de F#.

3.1.4.1 Prise en charge dans un cadre faiblement typé

Objective Caml représente les modules comme des blocs comprenant les valeurs globales des modules (notamment les fermetures). Les foncteurs de premier ordre sont implantés comme des fonctions sur ces blocs, et ainsi de suite : la fonctorisation est tout simplement traduite par l'abstraction fonctionnelle et l'application de foncteur par l'application fonctionnelle.

Nous avons choisi de réutiliser directement cette implantation, sans retoucher au code produit par Caml pour traiter les modules, foncteurs et applications de foncteurs. Cela nous permet d'éviter la voie délicate de la défonctorialisation, un sujet de thèse à part entière (voir à ce sujet l'article [76]).

Dans un cadre faiblement typé, c'est-à-dire sans génération de types utilisateur avec le code compilé (ce qui est le cas lorsque OCamIL reconstruit les types), la prise en charge du système de modules ne pose aucun problème. Les modules sont représentés comme des blocs génériques et il n'y a pas d'interférence avec le système de types CTS.

3.1.4.2 Prise en charge dans un cadre fortement typé

Les choses se compliquent lorsque des types utilisateur sont produits dans le code compilé, typiquement les classes qui implantent les types algébriques quand OCamIL propage les types. Détaillons la prise en charge des modules et des foncteurs dans ce cadre.

Types et espaces de noms introduits par les modules. Les types définis par l'utilisateur dans son code source peuvent figurer dans des sous-modules. Il convient de trouver un découpage dans le CTS qui reproduit cette structuration. Remarquons tout d'abord que le code intermédiaire généré par Caml ne reflète pas tout à fait l'agencement des modules. Il n'y a pas de notion d'espaces de noms ; tout est à plat dans le code d'un module car :

- Au niveau des valeurs, il n'y a aucune contrainte de nommage. Au sein d'une même unité de compilation, il est possible d'avoir plusieurs valeurs de même nom. Le fait qu'elles soient définies ou non dans un même sous-module n'a

pas d'importance : un générateur d'identificateurs uniques se charge de les différencier au niveau du code intermédiaire.
- Quant aux types, ils doivent avoir des noms uniques au sein d'un même sous-module. Si cela est pris en compte par le typeur, il n'y en a plus aucune trace dans le code engendré puisque celui-ci n'est pas typé.

Afin de distinguer des types homonymes définis dans des sous-modules différents, OCamIL utilise les espaces de noms du CTS, ce qui est valable à la fois pour les déclarations et les références de types Caml nommés. Ainsi un type Caml `t` (par exemple un enregistrement) défini dans le sous-module `A.B` d'un fichier d'implantation `M` provoque la déclaration d'une classe `M.A.B.t` en CIL. La même convention est utilisée lorsque les sous-modules Caml concernés sont des foncteurs.

Il n'y a pas de classe déclarée pour les types présents dans les signatures en arguments des foncteurs (dans leur déclaration ou leur application) car nous nous restreignons au cas où ces types sont abstraits, comme discuté ci-dessous.

Limitation des types CTS engendrés. L'objectif premier des types utilisateurs générés par OCamIL est l'amélioration des performances par l'optimisation des structures de données. Les définitions de types algébriques peuvent intervenir dans des modules (y compris fonctoriels) et des signatures de modules. Ce dernier cas pose problème lorsqu'il est exploité dans un argument de foncteur ou d'application de foncteur : nous allons voir qu'il n'est plus possible dans ces cas d'utiliser les représentations fines des types et qu'il est nécessaire de revenir à des implantations génériques moins performantes et moins lisibles.

Considérons les deux exemples suivants :

<table>
<tr>
<td><pre>module F (P:sig
             type t
             val test: t -> bool
             val t0: t
            end) = struct
   let f x = not (test x)
end</pre></td>
<td><pre>module F’ (P:sig
              type t = {x:int}
              val test: t -> bool
              val t0: t
             end) = struct
   let f x = not (test x)
end</pre></td>
</tr>
</table>

Pour le module `F` nous n'avons pas le choix : le type `t` est abstrait et il lui correspondra le type `object`. La fonction `f` sera implantée par une méthode de signature `bool f(object)`. Si on définit les modules :

```
module Param = struct
   type t = {x:int}
   let test v = v.x = 0
   let t0 = {x=0}
end

module M = F(Param)
module M’ = F’(Param)
```

L'application `M.f (Param.t0)` ne pose pas de problème de typage : bien que de type `Param.t`, la valeur `t0` est stockée dans les variables globales sous le type `object`. Son passage à la fonction `Param.test` se fait via l'application générique `apply` opérant sur les objets (du fait que `Param.test` est extrait de ces mêmes variables globales sous un type générique il n'y a pas d'application directe). Le retour est transtypé en booléen, et finalement subit une négation dans le code de `F.f`.

Qu'en est-il de l'application `M'.f (Param.t0)` si le type `t` y est implanté par une classe utilisateur notée `Ct` ? La fonction `f` est implantée par une méthode de signature `bool f(class Ct)` alors que de son côté `Param.test` a pour signature `bool test(class Param.t)`. Lors de l'application on a un conflit entre les deux types différents `Param.t` et `Ct`.

Dans un schéma de compilation séparée on ne peut pas résoudre cette difficulté par l'introduction d'une relation d'héritage ou d'implantation d'interface (et donc de sous-typage dans le CTS) entre ces deux types, car ils peuvent ne pas avoir connaissance l'un de l'autre au moment de finaliser leur émission dans le code.

En conclusion il faut renoncer à la génération de types ad-hoc dans les signatures appliquées aux paramètres de foncteurs, faisant comme si tous les types définis ou redéfinis dans celles-ci sont abstraits. Les définitions de types abstraits ou par alias ne posent pas de difficulté. Pour ce qui est de la génération de types utilisateur dans des espaces de noms :

- À un sous-module non fonctoriel correspond un sous-espace de noms ; les types définis dans le sous-module figurent dans ce sous-espace.
- Les modules fonctoriels `functor (M1:S1)...(Mn:Sn) -> (M:S)` sont ramenés au cas où les signatures `S1...Sn` n'induisent pas de définition de type CTS : seules les définitions de types dans `M` vont provoquer des déclarations de classes CTS. Pour cela nous faisons correspondre au module résultat `M` un sous-espace de nom pour contenir ces définitions de types.
- Une application de foncteur n'introduit aucune définition de types autre que des alias : nul besoin de définir de sous-espace de noms ni d'engendrer des types.

Le cas des Generics. Bien qu'étant une extension très intéressante de la plate-forme pour la compilation de structures de données et des fonctions polymorphes, nous montrons ici que les Generics n'apportent pas de solution idéale au problème de l'implantation des foncteurs. Considérons le foncteur suivant :

```
module MakeSortable (P:sig
                        type t
                        val compare: t -> t -> bool
                    end) = struct
    let sort l =
    (* implantation d'un tri basé sur P.compare *)
end
```

alors l'implantation naturelle sous forme de classe paramétrée dans les Generics, visant à implanter l'instanciation du foncteur au moyen de l'instanciation de type, aura la forme (on utilise une notation C#) :

```
class MakeSortable<T> {
   public static List<T> sort<P> (List<T> l) where P:IComparable<T> {
     /* fait des appels à P.compare(T x, T y) */
   }
}
```

avec la nécessité de déclarer une interface générique supplémentaire :

```
interface IComparable<T> {
   bool compare(T x, T y);
}
```

En effet il n'est pas possible d'avoir des paramètres de types contraints de manière purement structurelle, ce qui impose de définir l'interface `IComparable<T>`. On voit sur cet exemple simple une limitation directe à la modularité puisque les candidats potentiels à l'instanciation de `MakeSortable<T>.Sort<P>` doivent tous déclarer qu'ils implantent l'interface `IComparable<T>` a priori !

3.2 Production et exécution du code

Nous décrivons ici les dernières étapes de la chaîne de compilation : tout d'abord la transformation de la représentation `Ctypedlambda` vers une représentation Caml du code CIL, et ensuite la génération et l'édition de liens d'assemblages .NET contenant ce code.

3.2.1 De `Ctypedlambda` à `IL`

Le back-end de OCamIL, prenant la suite des étapes illustrées sur la figure 2.2 et indépendante du mode d'annotation de types utilisé, se présente de la manière suivante :

$$\texttt{typeinfo Ctypedlambda} \xrightarrow{\texttt{ILMcompile}} \texttt{ILM} \xrightarrow{\texttt{CompIL}} \texttt{IL} \xrightarrow{\texttt{EmitIL}} \text{code-octet}$$

Nous détaillons pour commencer les représentations intermédiaires `ILM` et `IL` figurées ci-dessus et montrons ensuite le passage de `typeinfo Ctypedlambda` à celles-ci.

3.2.1.1 Les représentations intermédiaires `ILM` et `IL`

Code `ILM`. Le langage `ILM` nous sert d'intermédiaire entre les deux représentations `Ctypedlambda` et `IL`. On peut voir `ILM` comme un langage de macros sur `IL`, le langage suivant dans la chaîne de compilation (décrit plus bas).

Les expressions M sont définies de la manière suivante :

M := $\textsc{Arg}$(x: t) *(variable à trouver parmi les arguments de la méthode)*
| $\textsc{Loc}$(x: t) *(variable à trouver parmi les variables locales)*
| $\textsc{Let}$(x: t,M_1,M_2)
| n *(constante entière)*
| $\textsc{Gapply}(M,M_1,\ldots,M_n)$
| $\textsc{Dapply}(L,M_1,\ldots,M_n)$
| $\textsc{If}(M,M_1,M_2)$
| $\textsc{Switch}(M{:}\ \mathtt{t},i_1 \to M_1,\ldots,i_n \to M_n,\_ \to M_d)$
| $\textsc{Catch}_i(x_1{:}\ \mathtt{t_1},\ldots,x_n{:}\ \mathtt{t_n},M_1,M_2)$
| $\textsc{Fail}_i(M_1,\ldots,M_n)$
| $\textsc{Prim}(P,M_1,\ldots,M_n)$
| $\{M\}_{\mathtt{t_1}\to\mathtt{t_2}}$ *(transtypage)*

Les types t correspondent aux types CTS :

t := `tref` *cid* | `int` | `bool` | t[] (*cid* désigne un nom de classe CTS pleinement qualifié par les espaces de noms).

Certains types `tref` *cid* prédéfinis par le runtime OCamIL seront écrits dans la suite de manière plus concise : `object`, `clos` et `sclos` (fermetures OCamIL simple et partagée).

Les primitives P sont proches de celles définies pour `Clambda` mais la gestion des blocs et des fermetures est plus explicite :

P := + | - | * | / *(opérations arithmétiques)*
| == | != *(comparaisons)*
| $\textsc{MkTop}_{\mathtt{cid}}$ *(création d'une fermeture sans environnement)*
| $\textsc{MkEnv}_{\mathtt{cid}}$ *(création d'une fermeture avec environnement)*
| $\textsc{MkSclos}_{\mathtt{cid}}$ *(création d'une fermeture partagée)*
| $\textsc{MkMrec}_{\mathtt{cid}}$ *(création d'un membre d'une fermeture partagée)*
| $\textsc{MkObject}_{\mathtt{cid}}$ *(création d'un objet quelconque)*
| $\textsc{GetFld}_{\mathtt{fref:t_r}}$ *(accès au champ d'une classe)*

Dans ces notations, *cid* désigne un nom de classe CTS et $fref$ est une référence de champ (c'est la donnée d'un nom de classe CTS qualifié et d'une chaine de caractères nommant le champ). Les primitives réelles de `ILM` sont données en annexe.

En plus d'un langage de termes, `ILM` définit l'organisation du code compilé pour un module Caml en différentes classes et méthodes, et dresse la liste des types CTS définis pour les besoins de cette compilation.

Ainsi une unité de compilation `ILM` est donnée par (FS,FP,V,R) où FS est une liste de fermetures simples, FP est une liste de fermetures partagées, V une liste de variants et R une liste d'enregistrements.

De plus, chaque fermeture simple est donnée par (L,cid,sig,a,$flds$,M) où :
- L est l'étiquette de la fermeture,

- *cid* est le nom de la classe CTS qui va implanter la fermeture,
- *sig* est la signature de la fonction exprimée au moyen de types du CTS (on écrira $sig = \mathtt{t_1} \to \ldots \to \mathtt{t_n} \to \mathtt{t_r}$) et a son arité,
- *flds* est une liste de champs typés (chaque champ typé est la donnée d'un couple $fref : \mathtt{t}$, c'est-à-dire une référence de champ et un type CTS),
- M est un terme de la grammaire détaillée ci-dessus qui implante la fonction de la fermeture simple.

L'expression toplevel d'un module prend la forme d'une fermeture simple conventionnelle (d'étiquette $\top$).

Chaque fermeture partagée est donnée par $(cid,flds)$: son nom de classe CTS et la description de ses champs.

V est formée d'une liste de couples (`cid`,`dv`) conservant la description de chaque variant défini dans l'unité de compilation avec le nom de type CTS choisi pour le représenter. R est formée d'une liste de couples (`cid`,`dr`) qui suit le même principe.

La compilation d'un module Caml, qui à l'entrée de la phase terminale de la chaîne se résume à une expression complexe $\mathbf{u_M}$ du langage `Ctypedlambda` conduit à la génération d'une unité de compilation `ILM` (FS,FP,V,R) dans laquelle se répartissent les sous-expressions de $\mathbf{u_M}$ (voir la section 3.2.1.2).

Code `IL`. C'est le dernier langage intermédiaire avant émission du code-octet sur le disque et c'est donc l'objectif final de la chaîne de compilation de OCamIL. Le langage `IL` est tout simplement une représentation Caml du code-octet CIL comportant en particulier les définitions et références de types, les types valeurs et les types références, et les méta-données.

De plus le langage `IL` recèle la structure complète générée pour un module Caml, avec :
- l'organisation du module compilé en espaces de noms, classes, champs et méthodes,
- pour chaque méthode : prototype, variables locales et suite d'instructions CIL.

Notations. Nous ne formalisons pas ici le langage `IL`. Il suffit de signaler que dans la suite une séquence d'instructions sera notée $I_1 ; \ldots ; I_n$ (où chaque instruction est une instruction CIL parmi celles données en annexe). Il est possible de donner des étiquettes à des instructions, que l'on présentera en indices dans la séquence. Par exemple dans $I_1 ;_L I_2 ; \ldots$ l'étiquette L désigne la deuxième instruction I_2 alors que dans $I_1 ; I_2 ;_L \ldots$ elle désigne l'instruction suivante.

L'émission de code réel à partir de la représentation `IL` est une sérialisation directe au format physique des fichiers PE (la section 3.2.2 donne plus de détails sur l'émission de code objet et l'édition de liens). Le véritable enjeu est donc de parvenir à la représentation `IL`.

3.2.1.2 Compilation de `Ctypedlambda` en code ILM

Le passage de `Ctypedlambda` à `ILM` est l'étape la plus importante de la phase finale de OCamIL, puisqu'il met en œuvre :
- le choix effectif des représentations, en termes de types CTS et des primitives les manipulant,
- le découpage du code dans la logique de la plate-forme visée.

Gestion des types. Trois opérations principales sur les types ont lieu au niveau de `ILM` : la traduction de types `typeinfo` en références de types CTS, l'insertion des transtypages $\{\mathrm{M}\}_{\mathtt{t_1}\rightarrow\mathtt{t_2}}$ et la définition des types CTS générés par la compilation du module source :

- Il n'y a pas de difficulté particulière à traduire les types `typeinfo` en références de types CTS. Toutefois cette transformation peut être paramétrée par des arguments en ligne de commande du compilateur OCamIL : il est ainsi possible par exemple de choisir de représenter les chaînes de caractères (type `string`) de différentes manières, si bien que le type visé pourra être `string`, `tref System.StringBuilder` ou `char[]`. De même les types enregistrements et variants peuvent être envoyés sur leur représentations naturelles sous forme de classes dédiées mais aussi vers des blocs génériques `object[]`.
- Les primitives de transtypage $\{\mathrm{M}\}_{\mathtt{t_1}\rightarrow\mathtt{t_2}}$ sont insérées lors de la compilation des expressions `Ctypedlambda` (voir une présentation formelle de cette transformation ci-après).
- Dans le cadre de la propagation de types, les types algébriques susceptibles d'être implantés par des classes sont listés dans la variable `camil_typedecls` décrite à la section 2.2.4. Son contenu est parcouru afin de générer les entrées correspondantes dans les listes V et R d'une unité de compilation `ILM`.

Remarque : les types algébriques Caml peuvent très bien n'être définis qu'au niveau de fichiers d'interfaces `.mli` sans fichier d'implantation correspondant. Dans ce cas, la compilation du fichier d'interface provoque la génération d'un fichier objet non trivial contenant des déclarations de classes CTS. En conséquence, les fichiers objets provenant d'interfaces sans implantation associée et contenant des types algébriques doivent être passés à l'éditeur de liens OCamIL alors que ce n'est pas le cas pour le compilateur Caml standard. Voilà une différence qui peut notamment amener à retoucher des fichiers `Makefile` pour la recompilation d'un programme Caml avec OCamIL.

Afin de poursuivre notre formalisation sur un sous-ensemble du langage traité, nous utilisons la fonction θ qui plonge les types de la grammaire `typeinfo` dans les types CTS, définie comme suit :

$\theta(\mathsf{int}) = \mathtt{int}$
$\theta(\mathsf{bool}) = \mathtt{bool}$
$\theta(\mathsf{block}) = \mathtt{object}[]$
$\theta(\mathsf{record}_{\mathsf{pa,dr}}) = \mathtt{tref\ Cid}(\mathsf{pa})$
$\theta(\mathsf{variant}_{\mathsf{pa,dv}}) = \mathtt{tref\ Cid}(\mathsf{pa})$
$\theta(\mathsf{t}_1 \to \ldots \to \mathsf{t}_\mathsf{n} \to \mathsf{t}) = \mathtt{clos}$
$\theta(\mathsf{clos}) = \mathtt{clos}$
$\theta(\mathsf{sclos}) = \mathtt{sclos}$
$\theta(\mathsf{obj}) = \mathtt{object}$

La fonction `Cid` utilisée ci-dessus associe un nom de classe CTS à un chemin de type absolu pa.

La transformation précédente correspond au compilateur OCamIL utilisé avec des options de compilation standard, associant aux types algébriques propagés des représentations dédiées et aux autres blocs une représentation générique.

Compilation des termes `Ctypedlambda`. Une unité de compilation Caml est comprise dans une seule expression `Ctypedlambda` (appelée « expression toplevel »). La transformation de celle-ci vers la représentation `ILM` conduit à un découpage et une répartition de ses sous-expressions entre différentes méthodes, le produit de la transformation étant une unité de compilation `ILM` (FS,FP,V,R).

Nous ne nous intéressons ici formellement qu'à la production de termes `ILM` dans les fermetures simples FS que l'on peut présenter comme un ensemble de couples (étiquette de fonction, terme `ILM`).

Afin de simplifier l'écriture de cette transformation, nous avons recours à une notation spéciale : le résultat est exprimé comme un seul arbre de termes `ILM`, mais dont chaque nœud est décoré par une étiquette de méthode L (ce que nous notons $\overline{M}^L$). Si on projette l'arbre décoré sur une étiquette L_0 donnée, on obtient un sous-arbre qui est exactement le terme compilé pour le code de la fermeture étiquetée par L_0.

La transformation $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{u})$ d'un terme `Ulambda` vers `ILM` est paramétrée par :

- l'étiquette Lb d'une fermeture simple de l'unité de compilation `ILM` dont le code est en cours de transformation,
- le type t correspondant au type CTS anticipé pour la valeur laissée sur la pile suite à l'exécution des instructions engendrées (ce type est fourni par le contexte),
- un environnement de compilation $E = (Sig,A,L,O_c,O_t,op)$ détaillé ci-dessous.

L'environnement $E = (Sig,A,L,O_c,O_t,op)$ est formé de :

- Sig mémorise les signatures CTS des fonctions déjà définies, associées à leur étiquette. On notera $Sig(L) = \mathtt{t}_1 \to \ldots \to \mathtt{t}_\mathtt{n} \to \mathtt{t}_\mathtt{r}$.

- A (resp. L) mémorise les types des arguments (resp. variables locales) de la méthode en cours de compilation, associés à leur identifiant. On notera $A(\mathbf{x}) = \mathtt{t}$ (resp. $L(\mathbf{x}) = \mathtt{t}$).
- O_c et O_t décrivent tous les deux des *offsets* de fermeture. Ce sont des triplets $(\mathbf{x},\mathtt{cid},o)$ où $\mathbf{x}$ est un identifiant associé à la fermeture, `cid` le nom de la classe CTS engendrée pour l'implantation de la fermeture, et o est une fonction partielle définie sur les entiers qui, à chaque indice du bloc représentant la fermeture, associe la description du champ stockant la valeur correspondante. On notera $o(n) = (fref,\mathtt{t_f})$, où $fref$ est la référence au champ et $\mathtt{t_f}$ est le type CTS du champ. O_c désigne la fermeture courante (dont une des fonctions contient l'expression en cours de compilation) et O_t décrit une fermeture définie dans une expression *let*, en passe d'être affectée à des identifiants (voir les détails de la transformation dans la suite).
- *op* est un entier donnant la position de la fonction en cours de compilation au sein de la fermeture qui la définie (elle-même décrite par l'offset O_c). Cet entier vaut 0 pour une fermeture simple et n'est réellement utilisé que dans le cas de fonctions mutuellement récursives, où il indique la position du pointeur sur la fonction courante au sein de la fermeture partagée.

Notons que la fonction o faisant partie de la description des offsets de fermetures prend ses valeurs exactement sur les indices auxquels l'implantation de Caml associe des pointeurs de fonctions et des champs d'environnement.

La transformation utilise une fonction $\mathcal{P}$ de choix des primitives : à une primitive $\mathbf{\Pi}$ de `CtypedLambda`, une signature de `typeinfo` $\mathtt{t_1} \to \ldots \to \mathtt{t_n} \to \mathtt{t_r}$ et un type CTS de retour, elle associe une primitive `ILM` et sa signature CTS.

La transformation consiste à calculer $\mathfrak{U}^{\top}_{E,\mathtt{object}}(\mathbf{u_M})$ sur l'expression toplevel $\mathbf{u_M}$ avec $E = (Sig,A,L,O_c,O_t,0)$ l'environnement trivial (dont tous les éléments sont vides). Voici les différents cas de la transformation :

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{x}) = \{\overline{\textsc{Arg}}^{Lb}(\mathrm{X}\colon \mathtt{t_x})\}_{\mathtt{t_x}\to\mathtt{t}}$ si $A(\mathbf{x}) = \mathtt{t_x}$

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{x}) = \{\overline{\textsc{Loc}}^{Lb}(\mathrm{X}\colon \mathtt{t_x})\}_{\mathtt{t_x}\to\mathtt{t}}$ si $L(\mathbf{x}) = \mathtt{t_x}$

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{n}) = \{\overline{\mathrm{N}}^{Lb}\}_{\mathtt{int}\to\mathtt{t}}$

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{Dapply}(L,\mathbf{u_1},\ldots,\mathbf{u_n})) = \{\overline{\textsc{Dapply}}^{Lb}(L,\mathfrak{U}^{Lb}_{E,\mathtt{t_1}}(\mathbf{u_1}),\ldots,\mathfrak{U}^{Lb}_{E,\mathtt{t_n}}(\mathbf{u_n}))\}_{\mathtt{t_r}\to\mathtt{t}}$
  avec $Sig(L) = \mathtt{t_1} \to \ldots \to \mathtt{t_n} \to \mathtt{t_r}$

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{Gapply}(\mathbf{u},\mathbf{u_1},\ldots,\mathbf{u_n})) =$
  $$\{\overline{\textsc{Gapply}}^{Lb}(\mathfrak{U}^{Lb}_{E,\mathtt{clos}}(\mathbf{u}),\mathfrak{U}^{Lb}_{E,\mathtt{object}}(\mathbf{u_1}),\ldots,\mathfrak{U}^{Lb}_{E,\mathtt{object}}(\mathbf{u_n}))\}_{\mathtt{object}\to\mathtt{t}}$$

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\textbf{if u then u}_\mathbf{1}\textbf{ else u}_\mathbf{2}) = \overline{\textsc{If}}^{Lb}(\mathfrak{U}^{Lb}_{E,\mathtt{bool}}(\mathbf{u}),\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{u_1}),\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{u_2}))$

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{fail_i}(\mathbf{u_1}\colon \mathsf{t}_1,\ldots,\mathbf{u_n}\colon \mathsf{t_n})) = \overline{\text{FAIL}}^{Lb}_i(\mathfrak{U}^{Lb}_{E,\theta(\mathsf{t}_1)}(\mathbf{u_1}),\ldots,\mathfrak{U}^{Lb}_{E,\theta(\mathsf{t_n})}(\mathbf{u_n}))$

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{catch_i}\ \mathbf{u_1}\ \mathbf{with}\ \mathbf{x_1}\colon \mathsf{t}^{\mathsf{x}}_1\ldots \mathbf{x_n}\colon \mathsf{t}^{\mathsf{x}}_{\mathsf{n}} \rightarrow \mathbf{u_2}) =$
$$\overline{\text{CATCH}}^{Lb}_i(\mathrm{X}_1{:}\theta(\mathsf{t}^{\mathsf{x}}_1),\ldots,\mathrm{X}_n{:}\theta(\mathsf{t}^{\mathsf{x}}_{\mathsf{n}}),\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{u_1}),\mathfrak{U}^{Lb}_{E',\mathtt{t}}(\mathbf{u_2}))$$

où $E' = (Sig,A,L',O_c,O_t,op)$ est un environnement qui prolonge E par la définition de nouvelles variables locales : $L'(\mathrm{X}_i) = \theta(\mathsf{t_i})$ et est identique à L sinon.

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{switch}\ \mathbf{u}\colon \mathsf{t}\ \mathbf{with} \begin{cases} \mathbf{i_1} \rightarrow \mathbf{u_1} \\ \vdots \\ \mathbf{i_n} \rightarrow \mathbf{u_n} \\ \_ \rightarrow \mathbf{u_d} \end{cases}) =$
$$\overline{\text{SWITCH}}^{Lb}(\mathfrak{U}^{Lb}_{E,\theta(\mathsf{t})}(\mathbf{u}){:}\theta(\mathsf{t}),i_1 \rightarrow \mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{u_1}),\ldots,i_n \rightarrow \mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{u_n}),\_ \rightarrow \mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{u_d}))$$

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{offset_n}(\mathbf{x})) = \{\overline{\text{PRIM}}^{Lb}(\text{GETFLD}_{\mathtt{fref:t_f}},\mathfrak{U}^{Lb}_{E,\mathtt{tref\ cid}}(\mathbf{x}))\}_{\mathtt{t_f}\rightarrow\mathtt{t}}$
lorsque $\mathbf{x}$ est consigné dans O_c, et que $O_c = (\mathbf{x},\mathtt{cid},o)$ avec $o(n{+}op) = (fref,\mathtt{t_f})$.

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{offset_n}(\mathbf{x})) = \{\overline{\text{PRIM}}^{Lb}(\text{GETFLD}_{\mathtt{fref:t_f}},\mathfrak{U}^{Lb}_{E,\mathtt{tref\ cid}}(\mathbf{x}))\}_{\mathtt{t_f}\rightarrow\mathtt{t}}$
lorsque $\mathbf{x}$ est consigné dans O_t, et que $O_t = (\mathbf{x},\mathtt{cid},o)$ avec $o(n) = (fref,\mathtt{t_f})$.

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{prim}(\mathbf{Field_n},\mathbf{x})) = \{\overline{\text{PRIM}}^{Lb}(\text{GETFLD}_{\mathtt{fref:t_f}},\mathfrak{U}^{Lb}_{E,\mathtt{tref\ cid}}(\mathbf{x}))\}_{\mathtt{t_f}\rightarrow\mathtt{t}}$
lorsque $\mathbf{x}$ est consigné dans O_c, et que $O_c = (\mathbf{x},\mathtt{cid},o)$ avec $o(n{+}op) = (fref,\mathtt{t_f})$.

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{let}\ \mathbf{x} = \mathbf{closure}(\ldots)\colon \mathsf{t}_1\ \mathbf{in}\ \mathbf{u_2}\colon \mathsf{t}_2) =$
$$\overline{\text{LET}}^{Lb}(\mathrm{X}\colon \mathtt{tref\ cid},\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{closure}(\ldots)),\mathfrak{U}^{Lb}_{E',\mathtt{t}}(\mathbf{u_2}))$$

où la compilation de la fermeture produit une description d'offset de la forme $(\mathbf{env},\mathtt{cid},o')$ pour n fermetures mutuellement récursives (cela prend aussi en compte le cas $n = 1$), d'étiquettes L_i et de signature $\mathtt{t}^1_{\mathtt{i}} \rightarrow \ldots \rightarrow \mathtt{t}^{\mathtt{m_i}}_{\mathtt{i}} \rightarrow \mathtt{t_i}$ (pour $1 \leq i \leq n$) ;

et $E' = (Sig',A,L',O_c,O'_t,op)$ est un environnement qui prolonge E par la définition d'une nouvelle variable locale, un nouvel offset temporaire et un ajout aux signatures :

- $L'(\mathrm{X}) = \mathtt{tref\ cid}$ et est identique à L sinon.
- $O'_t = (\mathbf{x},\mathtt{cid},o')$.
- $Sig'(L_i) = \mathtt{t}^1_{\mathtt{i}} \rightarrow \ldots \rightarrow \mathtt{t}^{\mathtt{m_i}}_{\mathtt{i}} \rightarrow \mathtt{t_i}$ (pour $1 \leq i \leq n$), et est identique à Sig sinon.

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{let}\ \mathbf{x} = \mathbf{u_1}\colon \mathsf{t}_1\ \mathbf{in}\ \mathbf{u_2}\colon \mathsf{t}_2) = \overline{\text{LET}}^{Lb}(\mathrm{X}{:}\theta(\mathsf{t}_1),\mathfrak{U}^{Lb}_{E,\theta(\mathsf{t}_1)}(\mathbf{u_1}),\mathfrak{U}^{Lb}_{E',\mathtt{t}}(\mathbf{u_2}))$
où $E' = (Sig,A,L',O_c,O_t,op)$ est un environnement qui prolonge E par la définition d'une nouvelle variable locale : $L'(\mathrm{X}) = \theta(\mathsf{t}_1)$ et est identique à L sinon.

- $\mathfrak{U}^{Lb}_{E,\mathtt{t}}(\mathbf{closure} \begin{pmatrix} \mathbf{L_1},\mathbf{a_1},\mathbf{x^1_1}\colon \mathsf{t}^1_1\ldots \mathbf{x^{k_1}_1}\colon \mathsf{t}^{\mathsf{k_1}}_1,\mathbf{u_1}\colon \mathsf{t}_1 \\ \vdots \\ \mathbf{L_n},\mathbf{a_n},\mathbf{x^1_n}\colon \mathsf{t}^1_{\mathsf{n}}\ldots \mathbf{x^{k_n}_n}\colon \mathsf{t}^{\mathsf{k_n}}_{\mathsf{n}},\mathbf{u_n}\colon \mathsf{t_n} \end{pmatrix} (\mathbf{u'_1}\colon \mathsf{t}'_1,\ldots,\mathbf{u'_m}\colon \mathsf{t}'_{\mathsf{m}}))$
(cas $n > 1$)

$= \{\overline{\text{PRIM}}^{Lb}(\overline{\text{MKSCLOS}}^{Lb}_{cid}, M_1, \ldots, M_n, \mathfrak{U}^{Lb}_{E,\theta(\mathsf{t}'_1)}(\mathbf{u'_1}), \ldots, \mathfrak{U}^{Lb}_{E,\theta(\mathsf{t}'_m)}(\mathbf{u'_m}))\}_{(\texttt{tref cid})\to\texttt{t}}$

où pour tout $1 \leq i \leq n$:

- $M_i = \overline{\text{PRIM}}^{Lb}(\overline{\text{MKMREC}}^{Lb}_{cid_i}, \mathfrak{U}^{L_i}_{E_i,\theta(\mathsf{t_i})}(\mathbf{u_i}))$ lorsque $\mathbf{a_i} = \mathbf{k_i} - \mathbf{1}$
- $M_i = \overline{\text{PRIM}}^{Lb}(\overline{\text{MKTOP}}^{Lb}_{cid_i}, \mathfrak{U}^{L_i}_{E_i,\theta(\mathsf{t_i})}(\mathbf{u_i}))$ lorsque $\mathbf{a_i} = \mathbf{k_i}$

où l'on utilise des nouveaux environnements $E_i = (Sig, A'_i, L, O'_c, O_t, op'_i)$ tels que :

- A'_i remplace totalement A en définissant $A'_i(\mathbf{x_i^j}) = \mathtt{t_i^j}$ pour tout $1 \leq j \leq k_i$.
- op'_i est la position de la ième fonction dans la fermeture partagée.
- O'_c est une description de la fermeture : $O'_c(o_1) = (\texttt{tref cid}_1), \ldots, O'_c(o_n) = (\texttt{tref cid}_n)$ et $O'_c(o_{n+1}) = \theta(\mathsf{t}'_1), \ldots, O'_c(o_{n+m}) = \theta(\mathsf{t}'_m)$.

Dans ce qui précède cid est le nom de la classe implantant la fermeture partagée et cid_i celui de la classe implantant la ième fermeture.

- $\mathfrak{U}^{Lb}_{E,\mathsf{t}}(\mathbf{closure}(\mathbf{L},\mathbf{a},\mathbf{x^1}\colon \mathsf{t}^1 \ldots \mathbf{x^k}\colon \mathsf{t}^k, \mathbf{u}\colon \mathsf{t})(\mathbf{u'_1}\colon \mathsf{t}'_1, \ldots, \mathbf{u'_m}\colon \mathsf{t}'_m)) =$
  - $\overline{\text{PRIM}}^{Lb}(\overline{\text{MKENV}}^{Lb}_{cid}, \mathfrak{U}^{Lb}_{E,\theta(\mathsf{t}'_1)}(\mathbf{u'_1}), \ldots, \mathfrak{U}^{Lb}_{E,\theta(\mathsf{t}'_m)}(\mathbf{u'_m}), \mathfrak{U}^{L}_{E',\theta(\mathsf{t})}(\mathbf{u}))$ lorsque $\mathbf{a} = \mathbf{k} - \mathbf{1}$
  - $\overline{\text{PRIM}}^{Lb}(\overline{\text{MKTOP}}^{Lb}_{cid}, \mathfrak{U}^{L}_{E',\theta(\mathsf{t})}(\mathbf{u}))$ lorsque $\mathbf{a} = \mathbf{k}$

où $E' = (Sig, A', L, O'_c, O_t, 0)$ tel que :

- A' remplace totalement A en définissant $A'(\mathbf{x^j}) = \mathtt{t^j}$ pour tout $1 \leq j \leq k$.
- O'_c est une description de la fermeture : $O'_c(o_1) = (\texttt{tref cid})$ et $O'_c(o_2) = \theta(\mathsf{t}'_1), \ldots, O'_c(o_{1+m}) = \theta(\mathsf{t}'_m)$.

Dans ce qui précède cid est le nom de la classe implantant la fermeture.

- $\mathfrak{U}^{Lb}_{E,\mathsf{t}}(\mathbf{prim}(\mathbf{\Pi}, \mathbf{u_1}\colon \mathsf{t}_1, \ldots, \mathbf{u_n}\colon \mathsf{t}_n)\colon \mathsf{t}_r) = \{\overline{\text{PRIM}}^{Lb}(\Pi', \mathfrak{U}^{Lb}_{E,\mathsf{t}'_1}(\mathbf{u_1}), \ldots, \mathfrak{U}^{Lb}_{E,\mathsf{t}'_n}(\mathbf{u_n}))\}_{\mathsf{t}'_r\to\mathsf{t}}$

où on a $\mathcal{P}(\mathbf{\Pi}, \mathsf{t}_1, \ldots, \mathsf{t}_n, \mathsf{t}_r, \mathsf{t}) = (\Pi', \mathsf{t}'_1, \ldots, \mathsf{t}'_n, \mathsf{t}'_r)$.

Remarque. La transformation $\mathbf{u} \to \mathfrak{U}^{Lb}_{E,\mathsf{t}}(\mathbf{u})$ maintient un invariant implicite : la séquence d'instructions CIL engendrée lors de l'étape suivante de compilation (expansion de `ILM` en code `IL`) agit sur la pile en ajoutant exactement une valeur de type `t`.

Détail d'implantation : le choix des primitives. Étant donné que les primitives Caml ne sont pas typées (voir la discussion de la section 2.2.2.2), il peut correspondre plusieurs primitives ILM à une seule primitive Caml. La sélection de la bonne primitive se fait précisément à l'aide des annotations de types `typeinfo`. C'est le rôle de la fonction $\mathcal{P}$ introduite de manière abstraite dans ce qui précède. La fonction réelle utilisée dans le compilateur OCamIL est :

```
select_prim: typeinfo -> Lambda.primitive * utypedlambda list
             -> ilmprim * ctstype list * ctstype
```

Celle-ci prend en entrée le sous-terme `Ctypedlambda` formé de la primitive (dont l'annotation de types `typeinfo` est dans un argument séparé) et de ses arguments typés. Elle retourne la primitive choisie avec sa signature complète exprimée en types CTS. Voyons l'action de `select_prim` à travers quelques exemples (on utilise les noms des primitives réelles détaillées en annexe) :

• Un cas simple : primitives `Pbintofint`, `Pintofbint` et `Pcvtbint` (on suppose dans cet exemple que les entiers 32 bits du CTS sont retenus pour l'implantation des entiers Caml).

Primitive d'origine	**Primitive choisie**	**Signature**
`Pbintofint bi`	`TPconvint` τ_{bi}	`int32` $\rightarrow \tau_{bi}$
`Pintofbint bi`	`TPconvint int32`	$\tau_{bi} \rightarrow$ `int32`
`Pcvtbint (bi1,bi2)`	`TPconvint` τ_{bi2}	$\tau_{bi1} \rightarrow \tau_{bi2}$

Avec τ_{bi} valant `int32`, `int64` ou `nint` selon la catégorie d'entier donnée par `bi`. Dans ce cas, ni l'annotation de type, ni les arguments de la primitive ne sont utilisés : elle contient elle-même suffisamment d'information.

• Création d'un bloc : primitive `Pmakeblock i (u1,...,un)`. Dans cet exemple, les types `list` et `option` sont représentés comme des variants classiques. Les types enregistrement et variant sont implantés à base de classes, et les autres blocs à base de tableaux d'objets (y compris les exceptions).

Type annoté	**Primitive choisie**	**Signature**

`TIblock`
`TItuple _`
`TIlazy _`
`TIexception`
	`Pmakeblock i`	`object` $\rightarrow \ldots \rightarrow$ `object` $\rightarrow$ `object[]`

`TIrecord id`	`TPbuildobject cid`	`t1` $\rightarrow \ldots \rightarrow$ `tn` $\rightarrow$ `tref cid`
`TIvariant id`	`TPbuildobject cid`	`t1` $\rightarrow \ldots \rightarrow$ `tn` $\rightarrow$ `tref cid`
`TIlist _`	Renvoyé sur `TIvariant list_id`	
`TIoption _`	Renvoyé sur `TIvariant option_id`	

Dans ce qui précède, les identificateurs `id` de variants et d'enregistrements permettent de retrouver la description complète des types en question et de constituer la définition de la classe `cid` donnée en paramètre de la primitive de construction d'objet `TPbuildobject`. D'autre part les types `t1`,...,`tn` désignent les types CTS correspondant directement aux annotations `typeinfo` incluses dans les arguments `u1`,...,`un`.

Il existe une autre primitive de création de blocs `Pmakearray at (u1...un)` utilisée principalement pour les tableaux (`at` indique le type de tableau, en particulier s'il s'agit d'un tableau de flottants) :

Type annoté	**Primitive choisie**	**Signature**
`TIarray _`	`Pmakearray at`	`object` $\rightarrow \ldots \rightarrow$ `object` $\rightarrow$ `CTobject[]`
`TIrecord id`	`TPbuildobject cd`	`t1` $\rightarrow \ldots \rightarrow$ `tn` $\rightarrow$ `tref cid`

Les enregistrements constitués uniquement de flottants sont optimisés de la même manière que les tableaux de flottants par le compilateur Caml, si bien qu'ils sont traités par les mêmes primitives.

• Écriture d'un champ de bloc : primitives `Psetfield` et `Psetfloatfield` (on peut noter que les types variant ne sont pas concernés par l'écriture car ils ne sont pas mutables).

Pour `Psetfield i u1 u2` (resp. `Psetfloatfield i u1 u2`) :

Type de `u1`	**Primitive choisie**	**Signature**
`TIblock`	`TPset_block i`	`object[]` → `object` → `void`
`TIrecord id`	`TPset_field fd`	`tref cid` → `tf` → `void`

La description du champ `fd` et son type `tf` sont obtenus à partir de `id` et de l'indice `i`.

• Lecture d'un champ de bloc : primitives `Pfield`, `Pfloatfield` ... et nouvelle primitive `Pfldtag`.
<u>Cas de `Pfield i u` et `Pfloatfield i u` (hors variants).</u>

Type de `u`	**Primitive choisie**	**Signature**

`TIblock`
`TItuple _`
`TIlazy _`
`TIexception`
	`TPget_block i`	`object[]` → `object`

`TIrecord id`	`TPget_field fd`	`tref cid` → `tf`

La description du champ `fd` et son type `tf` sont obtenus à partir de `id` et de l'indice `i` .

<u>Cas de `Pfldtag tg i u` pour les variants.</u> Les différents constructeurs d'un type variant sont associés à des classes différentes, si bien qu'il est nécessaire de connaître le tag du bloc qui est accédé en lecture afin de déterminer la primitive `TPget_field` adaptée. Or il n'est pas possible d'extraire cette information des annotations de types (qui ne distinguent pas les différents constructeurs d'un variant), ni des arguments. Heureusement, l'information manquante est connue statiquement au moment où la primitive `Pfield` est générée, c'est-à-dire au niveau du filtrage de motifs. Plutôt que de compliquer les systèmes de types `typing_annotation` et `typeinfo` pour qu'ils transmettent cette information, nous avons choisi plus simplement d'introduire une nouvelle primitive Caml `Pfldtag` : celle-ci est analogue à `Pfield` mis à part qu'elle transporte le tag en paramètre.

Type de `u`	**Primitive choisie**	**Signature**
`TIvariant id`	`TPget_field fd`	`tref cid` → `tf`
`TIlist _`	Renvoyé sur `TIvariant list_id`	
`TIoption _`	Renvoyé sur `TIvariant option_id`	

La description du champ `fd` et son type `tf` sont obtenus à partir de `id`, du tag `tg` et de l'indice `i`.

3.2.1.3 Génération de code-octet `IL`

Les expressions `ILM` sont simples et ne font plus figurer que des types CTS. Le passage à `IL` consiste simplement à expanser ces expressions en des suites d'instructions CIL.

L'émission d'instructions est fait avec un contrôle strict de la pile, modélisée par une liste de types, et des variables locales de chaque méthode. Le contrôle de pile nous a permis de déceler et de corriger rapidement des erreurs dans le code du compilateur. Les variables locales sont gérées de manière à économiser leur nombre : par exemple celles utilisées pour des variables Caml devenant hors de portée sont marquées comme libres. Bien sûr leur réutilisation est limitée par des contraintes de types.

Expansion des termes et primitives `ILM`. La traduction d'une unité de compilation (FS,FP,V,R) de `ILM` vers du code `IL` consiste à générer des classes (et dans celles-ci des champs et des méthodes) pour chaque fermeture simple de FS, chaque fermeture partagée de FP et chaque type algébrique de V et R.

<u>Génération de classes.</u> Nous ne la détaillons pas dans ce mémoire.

- Les classes d'implantations des enregistrements et des variants s'appuient sur un schéma bien défini. Les quelques particularités (champs contenant les données typées, constructeurs...) découlent facilement de la description des types enregistrements et variants.
- De même la génération des fermetures partagées consiste à définir une classe possédant de simples champs typés correspondant aux éléments de la fermeture (références sur les fermetures simples des différentes fonctions mutuellement récursives et environnement).
- Enfin les classes générées pour chaque fermeture simple (L,cid,sig,a,$flds$,M) s'appuient sur un squelette standard (constructeur, méthodes `exec` et `apply`). Le code de la méthode `apply` ne dépend que de la signature de la fonction (et de l'éventuelle présence de l'argument supplémentaire transmettant l'environnement). En revanche la méthode `exec` reçoit le code proprement dit, c'est-à-dire la transformation de M en suite d'instructions CIL. C'est cette transformation que nous détaillons dans la suite.

<u>Génération du code des fonctions.</u> Nous nous plaçons au niveau d'un terme `ILM` pour une fermeture simple donnée et nous formalisons la transformation qui aboutit à une suite d'instructions `IL`. La transformation utilise 1) des informations globales et 2) un environnement de compilation.

Les informations globales sont données par des fonctions arg, loc, fp, sig, ct :

- Pour chaque identifiant x , loc(x) (resp. arg(x)) retourne l'indice de x dans les variables locales (resp. arguments) de la méthode en cours de génération.

- fp est une fonction qui à chaque référence de types correspondant à une fermeture partagée, associe une description de cette fermeture en donnant 1) un indice dans les variables locales d'une variable supposée stocker une référence vers celle-ci et 2) la suite ordonnée de ses descripteurs de champs.
- ct et sig associent respectivement à chaque étiquette de fermeture le nom de classe CTS qui l'implante et la signature CTS de la fonction ; ces informations sont obtenues simplement à partir de la liste des fermetures simples FS de l'unité de compilation `ILM`.

Ces informations globales sont récupérées pour la fonction en cours de compilation au moyen d'une première passe sur le terme `ILM`[4] (spécifiquement pour rassembler les variables utilisées loc).

L'environnement de compilation est formé d'un triplet (w,f,sid) où :
- w est une sorte de continuation, pouvant prendre l'une des trois[5] valeurs suivantes : $wRet$ (fin de la méthode), $wCont$ (passage à l'instruction suivante) et $wBr\ L$ (saut vers l'étiquette L).
- f sert à la compilation des fail/catch : il donne pour chaque entier i la liste des indices des variables locales utilisées pour la propagation de valeurs d'un fail vers un catch, ainsi que d'une étiquette permettant de sauter vers le récupérateur de chaque catch. On notera $f(i) = (k_1, \ldots, k_n, L)$ si l'étiquette L est définie et $f(i) = (k_1, \ldots, k_n, ?)$ sinon.
- sid est la référence de types CTS correspondant à la fermeture partagée courante (elle peut ne pas être définie).

On associe à w une séquence d'instructions réalisant la continuation de la façon suivante : $W(wRet) = \texttt{ret}$, $W(wCont) =$ rien et $W(wBr\ L) = \texttt{br}\ L$.

La transformation $\mathfrak{M}_{w,f,sid}$ de `ILM` à `IL` est définie par cas comme suit (pour lancer la transformation on applique $\mathfrak{M}_{wRet,f_0,sid_0}$ au terme `ILM` complet d'une fermeture simple, où f_0 est triviale et sid_0 est indéfinie). Nous utilisons la syntaxe des instructions données en annexe.

- $\mathfrak{M}_{w,f,sid}(\mathrm{N}) = \texttt{ldc.i4 n}\ ;\ W(w)$

- $\mathfrak{M}_{w,f,sid}(\textsc{Arg}(\mathrm{X}\colon \texttt{t})) = \texttt{ldarg}\ arg(\mathrm{X})\ ;\ W(w)$

- $\mathfrak{M}_{w,f,sid}(\textsc{Loc}(\mathrm{X}\colon \texttt{t})) = \texttt{ldarg}\ loc(\mathrm{X})\ ;\ W(w)$

- $\mathfrak{M}_{w,f,sid}(\textsc{Let}(\mathrm{X}\colon \texttt{t},\mathrm{M}_1,\mathrm{M}_2)) = \mathfrak{M}_{wCont,f,sid}(\mathrm{M}_1)\ ;\ \texttt{stloc}\ loc(\mathrm{X})\ ;\ \mathfrak{M}_{w,f,sid}(\mathrm{M}_2)$

4. En réalité il n'y a qu'une seule passe dans le compilateur OCamIL, mais nous utilisons pour cela un environnement qui est modifié par effet de bord. Le présenter de cette façon dans le mémoire compliquerait inutilement.

5. Le compilateur OCamIL utilise une quatrième valeur $wLeave$: cette continuation indique qu'il faut sortir d'un bloc de traitement d'exceptions.

• $\mathfrak{M}_{w,f,sid}(\text{IF}(\text{M},\text{M}_1,\text{M}_2)) =$
$$\mathfrak{M}_{wCont,f,sid}(\text{M})\ ;\ \texttt{brfalse}\ L_F\ ;\ \mathfrak{M}_{w',f,sid}(\text{M}_1)\ ;_{L_F}\ \mathfrak{M}_{w,f,sid}(\text{M}_2)\ ;_{L_C}$$

avec $w' = w$ si $w \neq wCont$ et $w' = wBr\ L_C$ sinon.

• $\mathfrak{M}_{w,f,sid}(\text{SWITCH}(\text{M: }\texttt{t},i_1 \to \text{M}_1,\ldots,i_n \to \text{M}_n,\_ \to \text{M}_d)) =$
$$\mathfrak{M}_{wCont,f,sid}(\text{M})\ ;\ \texttt{ldfld int Variant} :: \texttt{tag}\ ;\ \texttt{switch}(L_{j_0},\ldots,L_{j_m})\ ;_{L_d}$$
$$\mathfrak{M}_{w',f,sid}(\text{M}_d)\ ;_{L_1}\ \mathfrak{M}_{w',f,sid}(\text{M}_1)\ ;\ldots;_{L_n}\ \mathfrak{M}_{w',f,sid}(\text{M}_n)\ ;_{L_C}$$

avec :
- $w' = w$ si $w \neq wCont$ et $w' = wBr\ L_C$ sinon.
- m est le maximum des i_k, et pour chaque $0 \leq l \leq m$, $\exists k/l = i_k \Rightarrow L_{j_l} = L_k$ et sinon $L_{j_l} = L_d$.

• $\mathfrak{M}_{w,f,sid}(\text{FAIL}_i(\text{M}_1,\ldots,\text{M}_n)) =$
$$\mathfrak{M}_{wCont,f,sid}(\text{M}_1)\ ;\ \texttt{stloc}\ k_1\ ;\ \ldots\ ;\ \mathfrak{M}_{wCont,f,sid}(\text{M}_n)\ ;\ \texttt{stloc}\ k_n\ ;\ br\ L$$

avec $f(i) = (k_1,\ldots,k_n,L)$.

• $\mathfrak{M}_{w,f,sid}(\text{CATCH}_i(\text{x}_1\text{: }\texttt{t}_1,\ldots,\text{x}_n\text{: }\texttt{t}_\texttt{n},\text{M}_1,\text{M}_2)) =$
$$\mathfrak{M}_{w',f',sid}(\text{M}_1)\ ;_L\ \mathfrak{M}_{wCont,f,sid}(\text{M}_2)\ ;_{L_C}$$

avec :
- $w' = w$ si $w \neq wCont$ et $w' = wBr\ L_C$ sinon.
- f' est défini de manière à indiquer l'étiquette L, c'est-à-dire que si $f(i) = (k_1,\ldots,k_n,?)$ on pose $f'(i) = (k_1,\ldots,k_n,L)$, f' étant identique à f pour tout entier $\neq i$.

• $\mathfrak{M}_{w,f,sid}(\text{GAPPLY}(\text{M},\text{M}_1,\ldots,\text{M}_n)) =$
$$\mathfrak{M}_{w,f,sid}(\text{GAPPLY}(\{\text{GAPPLY}(\text{M},\text{M}_1)\}_{\texttt{object}\to\texttt{clos}},\text{M}_2,\ldots,\text{M}_n)) \text{ pour } n \geq 2.$$

$\mathfrak{M}_{w,f,sid}(\text{GAPPLY}(\text{M},\text{M}_1)) =$
$$\mathfrak{M}_{wCont,f,sid}(\text{M})\ ;\ \mathfrak{M}_{wCont,f,sid}(\text{M}_1)\ ;$$
$$\texttt{call}_\texttt{inst}\ \texttt{object Closure} :: \texttt{apply(object)}\ ;\ W(w)$$

• $\mathfrak{M}_{w,f,sid}(\text{DAPPLY}(\text{L},\text{M}_1,\ldots,\text{M}_n)) =$
$$\mathfrak{M}_{wCont,f,sid}(\text{M}_1)\ ;\ \ldots\ ;\ \mathfrak{M}_{wCont,f,sid}(\text{M}_n)\ ;$$
$$\texttt{call}\ \texttt{t}_\texttt{r}\ ct(L) :: \texttt{apply}(\texttt{t}_1,\ldots,\texttt{t}_\texttt{n})\ ;\ W(w)$$

où $sig(L) = \texttt{t}_1 \to \ldots \to \texttt{t}_\texttt{n} \to \texttt{t}_\texttt{r}$.

• $\mathfrak{M}_{w,f,sid}(\{\text{M}\}_{\texttt{t}_1\to\texttt{t}_2}) = \mathfrak{M}_{wCont,f,sid}(\text{M})\ ;\ \mathfrak{C}(\texttt{t}_1,\texttt{t}_2)\ ;\ W(w)$

avec :

- $\mathfrak{C}(\texttt{t}_1,\texttt{t}_2) =$ rien si $\texttt{t}_1 = \texttt{t}_2$

- $\mathfrak{C}$(`tref C`$_1$,`tref C`$_2$) = `castclass C`$_2$
- $\mathfrak{C}$(`int`,`tref C`$_2$) = `box Int32` ; `castclass C`$_2$
- $\mathfrak{C}$(`tref C`$_1$,`int`) = `castclass Int32` ; `unbox Int32` ; `ldind.i4`

On trouvera des détails sur le transtypage dans la suite.

Transformation des primitives :

$\mathfrak{M}_{w,f,sid}(\text{PRIM}(+,\text{M}_1\text{M}_2)) = \mathfrak{M}_{wCont,f,sid}(\text{M}_1)$; $\mathfrak{M}_{wCont,f,sid}(\text{M}_2)$; `add` ; $W(w)$

$\mathfrak{M}_{w,f,sid}(\text{PRIM}(\text{-},\text{M}_1\text{M}_2)) = \mathfrak{M}_{wCont,f,sid}(\text{M}_1)$; $\mathfrak{M}_{wCont,f,sid}(\text{M}_2)$; `sub` ; $W(w)$

$\mathfrak{M}_{w,f,sid}(\text{PRIM}(*,\text{M}_1\text{M}_2)) = \mathfrak{M}_{wCont,f,sid}(\text{M}_1)$; $\mathfrak{M}_{wCont,f,sid}(\text{M}_2)$; `mul` ; $W(w)$

$\mathfrak{M}_{w,f,sid}(\text{PRIM}(/,\text{M}_1\text{M}_2)) = \mathfrak{M}_{wCont,f,sid}(\text{M}_1)$; $\mathfrak{M}_{wCont,f,sid}(\text{M}_2)$; `div` ; $W(w)$

$\mathfrak{M}_{w,f,sid}(\text{PRIM}(==,\text{M}_1\text{M}_2)) = \mathfrak{M}_{wCont,f,sid}(\text{M}_1)$; $\mathfrak{M}_{wCont,f,sid}(\text{M}_2)$; `ceq` ; $W(w)$

$\mathfrak{M}_{w,f,sid}(\text{PRIM}(!=,\text{M}_1\text{M}_2)) = \mathfrak{M}_{wCont,f,sid}(\text{M}_1)$; $\mathfrak{M}_{wCont,f,sid}(\text{M}_2)$; `ceq` ;
`ldc.i4.0` ; `ceq` ; $W(w)$

$\mathfrak{M}_{w,f,sid}(\text{PRIM}(\text{GETFLD}_{\texttt{fref:t}_\texttt{r}},\text{M})) = \mathfrak{M}_{wCont,f,sid}(\text{M})$; `ldfld t`$_\texttt{f}$ `fref` ; $W(w)$

$\mathfrak{M}_{w,f,sid}(\text{PRIM}(\text{MKBLOCK}_{tag},\text{M}_1,\ldots,\text{M}_n) =$
`ldc.i4` $n+1$; `newarr object` ;
`dup` ; `ldc.i4 0` ; $\mathfrak{M}_{wCont,f,sid}(\text{M}_1)$; `stelem.ref` ;
...
`dup` ; `ldc.i4` $n-1$; $\mathfrak{M}_{wCont,f,sid}(\text{M}_n)$; `stelem.ref` ;
`dup` ; `ldc.i4` n ; `ldc.i4` tag ; `box Int32` ; `stelem.ref` ; $W(w)$

$\mathfrak{M}_{w,f,sid}(\text{PRIM}(\text{MKOBJECT}_{\texttt{cid}},\text{M}_1,\ldots,\text{M}_n) =$
$\mathfrak{M}_{wCont,f,sid}(\text{M}_1)$; ... ; $\mathfrak{M}_{wCont,f,sid}(\text{M}_n)$; `newobj cid :: .ctor(t`$_1$`, ... ,t`$_\texttt{n}$`)` ; $W(w)$

où $t_1, \ldots t_n$ sont les types des champs $flds$ de la classe de R ou V dont la référence est cid.

$\mathfrak{M}_{w,f,sid}(\text{PRIM}(\text{MKTOP}_{\texttt{cid}}) =$ `newobj cid :: .ctor()` ; $W(w)$

$\mathfrak{M}_{w,f,sid}(\text{PRIM}(\text{MKENV}_{\texttt{cid}},\text{M}_1,\ldots,\text{M}_n) =$
$\mathfrak{M}_{wCont,f,sid}(\text{M}_1)$; ... ; $\mathfrak{M}_{wCont,f,sid}(\text{M}_n)$; `newobj cid :: .ctor(t`$_1$`, ... ,t`$_\texttt{n}$`)` ; $W(w)$

où $t_1, \ldots t_n$ sont les types des champs $flds$ de la fermeture simple dont la référence est cid.

$\mathfrak{M}_{w,f,sid}(\text{PRIM}(\text{MKSCLOS}_{\texttt{cid}},\text{M}_1,\ldots,\text{M}_n) =$

$\texttt{newobj cid} :: \texttt{.ctor}()$; $\texttt{stloc}\ i$;
$\texttt{ldloc}\ i$; $\mathfrak{M}_{wCont,f,cid}(\mathrm{M}_1)$; $\texttt{stfld}\ \texttt{t}_1\ \texttt{fref}_1$;
...
$\texttt{ldloc}\ i$; $\mathfrak{M}_{wCont,f,cid}(\mathrm{M}_n)$; $\texttt{stfld}\ \texttt{t}_\texttt{n}\ \texttt{fref}_\texttt{n}$;
$\texttt{ldloc}\ i$; $W(w)$

où $fp(cid) = (i,(\texttt{t}_1\ \texttt{fref}_1, \ldots, \texttt{t}_\texttt{n}\ \texttt{fref}_\texttt{n}))$

$\mathfrak{M}_{w,f,sid}(\textsc{Prim}(\textsc{MkMrec}_{\texttt{cid}}) = \texttt{ldloc}\ i$; $\texttt{newobj cid} :: \texttt{.ctor}(sid)$; $W(w)$

où $fp(sid) = (i, \ldots)$

Note : les types `t` ne sont pas utilisés pour cette transformation mais sont nécessaires à la bonne déclaration de la méthode (signature et types des variables locales), qui est prise en charge par une première passe sur le code.

Implantation des transtypages. Les transtypages sont mis en œuvre par les primitives `TPcast`. En voici quelques exemples :

Transtypage	**Code CIL**
`int32` → `object`	`box Int32`

`object` → `int32`	`castclass Int32`
	`unbox Int32`
	`ldind.i4`

`tref cid` → `object`	rien (voir la remarque ci-dessous)
`object` → `tref cid`	`castclass cid`
`string` → `tref StringBuilder`	`newobj StringBuilder::.ctor(string)`
`object` → `void`	`pop`

Remarques :
- Les transtypages d'un type référence vers un sur-type (comme ci-dessus le type variant vers le type objet) ne nécessitent aucune instruction CIL. L'algorithme de génération de code `IL` prend toutefois acte du changement de type pour sa gestion de pile.
- OCamIL optimise les transtypages enchaînés afin de n'engendrer que le minimum d'instructions.

3.2.2 Émission de code et édition de liens

3.2.2.1 Émission de code objet

La génération physique de code est une simple sérialisation de l'ultime représentation intermédiaire `IL` vers un fichier objet `.cmx` sur le disque dans lequel les références CTS seront résolues au moment de l'édition de liens. Au moment de cette sérialisation quelques optimisations sont introduites dans le code CIL : par exemple les instructions de branchement sont remplacées par leur version courte quand les dimensions du code le permettent.

Le format des fichiers objets dépend du type d'assembleur CIL utilisé ; nous en présentons quelques-uns dans la suite.

Avec assembleur externe. C'est la solution initiale mise en œuvre par OCamIL. L'idée est d'utiliser l'assembleur en ligne de commande du kit de développement de la plate-forme .NET, par exemple `ilasm.exe` pour la plate-forme Windows. Le code CIL est émis sous forme de code assembleur dans un fichier texte, qui est ensuite soumis à la compilation par la commande externe.

L'avantage est la rapidité d'implantation, qui repose sur la fiabilité de l'outil externe. En phase de développement initial, mieux vaut ne pas perdre de temps sur de problèmes bas-niveau de génération de code.

Cette approche a tout de même plusieurs inconvénients :

- Les performances ne sont pas bonnes, car il faut imprimer le code CIL à partir de sa représentation interne, pour ensuite repasser par la phase d'analyse syntaxique de `ilasm.exe`.
- Reposer sur une commande externe pose plusieurs problèmes : un problème d'installation et de configuration (il faut que le kit de développement .NET soit installé pour utiliser OCamIL, que la commande soit dans le chemin `$PATH`) et un problème de sécurité (à un niveau de sécurité standard, il n'est pas toujours autorisé d'appeler une commande externe, en particulier dans les appliquettes, voir section 5.2.2 pour plus de détails).
- Enfin, et ce n'est pas le moindre des défauts, l'utilisation de `ilasm.exe` trahit le principe de compilation séparée. En effet celui-ci ne produit que des assemblages fermés, ce qui n'est désiré qu'à l'étape d'édition de liens. Cela signifie en pratique que le code objet est stocké sous une forme non-compilée (directement du texte, ou encore la représentation interne de OCamIL persistée) et que la compilation a lieu une fois sur le produit de l'édition de liens. Cela implique une recompilation complète en cas de modification du moindre fichier d'implantation.

Avec assembleur interne. Une solution aux problèmes précédents serait d'utiliser un assembleur directement incorporé au compilateur OCamIL. Cela suppose de maîtriser le format physique des fichiers PE.

Avec la bibliothèque `Reflection`. Il est également possible d'utiliser la bibliothèque `Reflection` de l'API .NET. Celle-ci permet de générer dynamiquement des assemblages .NET, sur le disque ou en mémoire.

Les différents composants d'un assemblage (classes, méthodes, code CIL, etc. . .) sont reflétés dans des structures de données qu'il est possible de construire au moyen de cette API, pour ensuite les persister. Il est tout à fait possible d'utiliser cette approche dans un compilateur comme OCamIL.

Parmi les inconvénients :

- Cette approche ne peut fonctionner qu'avec le compilateur bootstrappé, seul à être en mesure d'utiliser l'API .NET (voir la section 5.2.1.1).
- L'interfaçage est délicat car il y a de nombreuses méthodes à utiliser, manipulant de nombreuses classes différentes. Cela peut être mis en œuvre au moyen d'outils dédiés à l'interopérabilité (la réalisation de ceci est détaillée à la section 5.2.2.1).
- Il y a un défaut commun avec le compilateur externe : il est compliqué de mettre en œuvre un réel mécanisme de compilation séparée, car on ne peut pas directement construire des méta-données pour des références externes non résolues.

Parmi les avantages :

- Il n'y a plus de problème possible lié au format physique (texte ou brut) du code puisqu'on travaille à travers une API.
- Cela élimine certains problèmes de sécurité : l'émission en mémoire ne met plus en jeu ni commande externe, ni même le disque dur, ce qui est particulièrement intéressant pour les appliquettes.

3.2.2.2 Édition de liens

L'édition de liens regroupe des fichiers objets et résout les références de types CTS afin de produire un assemblage .NET, sous forme d'un exécutable ou d'une bibliothèque. Il est possible de fournir un fichier `.snk` (*strong name key*) en ligne de commande à l'éditeur de liens OCamIL afin de donner un nom fort à l'assemblage produit.

La bibliothèque d'exécution `core_camil`. Tous les exécutables ou bibliothèques produits par OCamIL référencent la bibliothèque d'exécution `core_camil.dll`. Celle-ci contient dans l'espace de noms `CamIL` tous les types et méthodes nécessaires à l'exécution d'un programme Objective Caml sous .NET : la plupart de ces types ont été décrit précédemment. La bibliothèque standard de Caml est incluse dans cette bibliothèque, qui est donc suffisante pour faire tourner la plupart des programmes Caml.

L'assemblage `core_camil` est obtenu en compilant un mélange de code Caml (pour la bibliothèque standard, compilée avec OCamIL lui-même) et de code CIL écrit à la main. Les primitives externes utilisées par la bibliothèque standard (implantées en C dans le cas de Caml) référencent des méthodes incluses dans `core_camil` et implantées en CIL.

Résolution des références et assemblage du fichier produit. Un appel à l'éditeur de liens OCamIL en ligne de commande est de la forme :

```
ocamil [-a] -o fileout f1.cmx[a] f2.cmx[a] ... fn.cmx[a]
```

L'option `-a` indique de compiler une bibliothèque plutôt qu'un exécutable. Dans ce cas, en plus du fichier `fileout.dll` sera émis un fichier objet `fileout.cmxa`. Les fichiers objets peuvent être des `.cmx` contenant le code CIL d'un fichier d'implantation Caml ou des `cmxa` produits par la compilation de bibliothèques.

La résolution des dépendances dans un module donné se fait au sein des fichiers objets qui le précèdent dans la ligne de commande. Il s'agit pour chaque référence de type, de champ, de méthode, de trouver l'assemblage dans lequel il/elle est défini(e). Il y a trois possibilités :

- La définition se trouve dans la bibliothèque standard : l'assemblage recherché est `core_camil.dll`.
- La définition se trouve dans un fichier objet `cmx` : l'assemblage qualifiant la référence est l'assemblage courant, en cours de production.
- La définition est rapportée dans un fichier objet `cmxa` : l'assemblage est alors la bibliothèque associée au fichier objet[6].

Le traitement est différent pour les primitives externes (déclarations `external`) car elles contiennent explicitement la référence de l'assemblage comportant leur implantation. Cela permet d'appeler du code CIL qui n'est pas issu d'un module Caml et pour lequel il n'y a donc pas de fichier objet.

Le fichier produit incorpore le code des fichiers objets `cmx`. À chacun de ces modules physiques correspond un espace de nom dans lequel on trouve les classes générées (fermetures et types algébriques), éventuellement réparties entre différents sous-espaces de noms correspondant aux sous-modules. Les unités de compilation possèdent toutes une classe `Top` contenant le champ de leurs variables globales et une méthode `startup` qui procède à leur initialisation, c'est-à-dire à l'évaluation du code toplevel du module.

Au niveau de l'assemblage est ajoutée une classe de démarrage `MLTop` possédant une méthode `startup` procédant à l'initialisation de tous les objets formant l'unité empaquetée, dans l'ordre spécifié sur la ligne de commande. Celle-ci appelle la méthode `startup` de chaque module pour les `cmx` et son homologue sur chaque `dll` associée aux `cmxa`. La classe possède un champ booléen qui témoigne d'une initialisation effectuée : cela permet d'éviter l'initialisation multiple d'un assemblage dans un même compartiment d'exécution.

Les fichiers exécutables n'ont en plus des bibliothèques qu'une méthode `main` permettant de récupérer des arguments en ligne de commande et dont le code consiste à encadrer la méthode `startup` introduite ci-dessus par un gestionnaire d'exceptions

6. Moyennant l'utilisation de méta-données customisées, on pourrait résoudre ces références directement à l'examen de la `dll` et ainsi se passer du `cmxa`.

(visant à capturer toutes les exceptions non rattrapées dans le programme et à les afficher, comme le font les exécutables Caml classiques).

Liaison dynamique. L'implantation OCamIL adapte le module `Dynlink` à l'aide de la bibliothèque `Reflection`. Les composants pouvant être chargés dynamiquement sont des assemblages de bibliothèques (fichiers `dll` compilés par OCamIL). L'appel à `Dynlink.loadfile` charge l'assemblage en mémoire et exécute sa méthode `startup`.

Par ailleurs, une autre forme de liaison dynamique est utilisée pour l'implantation du toplevel OCamIL, que nous détaillons à la section 5.2.1.2.

Chapitre 4

Interopérabilité et OJacaré.NET

Nous examinons ici les techniques utilisées en matière d'interopérabilité.

Sommaire

4.1 Objective Caml et l'interopérabilité

Une critique récurrente des langages fonctionnels porte sur la difficulté de les interfacer avec d'autres langages de programmation. Autant le modèle des langages fonctionnels est bien accepté pour son élégance et son expressivité, autant le monde des langages fonctionnels est souvent considéré comme autiste ou du moins isolationniste. L'introduction de [38] met en garde :

> « *Programming languages that do not support a foreign-language interface die a slow, lingering death - good languages die more slowly than bad ones, but they all die in the end.* »

Cette critique a en fait été entendue et argumentée dans les articles généraux de Gabriel [41] et Wadler [89]. Plusieurs solutions ont été apportées pour des langages fonctionnels de caractéristiques différentes comme Lisp [69], SML [89] et Haskell [37].

Bien que de nombreux progrès aient été effectués dans l'interfaçage entre langages, certaines difficultés subsistent dans l'enchevêtrement de plusieurs mondes : systèmes de types incompatibles, valeurs hétérogènes, entrelacement des récupérateurs d'exceptions, gestionnaires mémoire et *multi-threading*. Ces traits sont liés aux langages abordés mais aussi aux plates-formes d'exécution de ceux-ci. De plus essayer d'adapter des traits de programmation existant dans un seul des deux langages interfacés peut faire perdre à l'ensemble un facteur important d'efficacité.

4.1.1 Les langages d'interface

L'interface d'un langage avec le monde extérieur est très souvent basée sur une interface de bas niveau avec la plate-forme d'exécution. Dans les langages fonctionnels elle correspond à l'appel d'une fonction externe (*Foreign Function Interface* ou *FFI*) et au passage d'arguments par copie pour les valeurs immédiates et par référence pour les valeurs structurées. Cette couche basse est ensuite utilisée pour construire une communication de plus haut niveau entre deux langages tant du point de vue du contrôle d'exécution que de la gestion mémoire, tout en garantissant la correction du typage. La richesse de ce modèle de communication permet d'utiliser de nouvelles bibliothèques et d'étendre un langage en intégrant de nouveaux traits de programmation.

On peut considérer deux familles de plates-formes : les bibliothèques d'exécution qui sont liées au code engendré par un compilateur pour produire un programme exécutable et les machines virtuelles gérant contrôle et mémoire. Ces bibliothèques d'exécution sont écrites en C, vu comme un assembleur de haut niveau, et offrent un mécanisme d'appel de fonctions externes C.

Pour un langage fonctionnel, la bibliothèque d'exécution intègre principalement une représentation des données dont les fermetures, un mécanisme général d'application, une gestion automatique de la mémoire et une récupération d'exception. Les difficultés d'interfaçage entre langages, reposant sur des bibliothèques d'exécution différentes, proviennent de l'entrelacement de contrôles issus de ces deux bibliothèques et de la construction de valeurs hétérogènes issues des deux représentations. La manipulation de valeurs fonctionnelles comme données des langages fonctionnels ajoute une difficulté supplémentaire.

4.1.1.1 Choix d'un langage d'interface

La construction d'une interface entre langages nécessite un choix de conception : comment et où décrit-on celle-ci?

- utiliser une bibliothèque d'appel extérieur réduite : l'encapsulation et le décodage des valeurs sont effectués par du code supplémentaire écrit à la main qui constitue une nouvelle bibliothèque C pouvant être appelée depuis Caml.
- utiliser les descriptions d'interfaces des modules/paquetages du langage externe pour engendrer automatiquement le code d'encapsulation : soit en intégrant les déclarations des interfaces du langage externe dans son langage, soit en utilisant un outil externe pour engendrer le code encapsulant. Dans ces deux cas, il est nécessaire de s'adapter aux déclarations du langage externe.
- définir un langage extérieur de description d'interface (IDL). Ce langage correspond à une intersection des modèles de programmation que l'on désire faire communiquer. De cette description le code peut être engendré automatiquement.

Ce choix de la technique à utiliser est effectué par rapport aux caractéristiques d'expressivité, de sûreté et de facilité d'emploi de l'interface tout en mesurant les coûts en efficacité que ces choix peuvent induire.

4.1.1.2 Problématiques de l'interface avec C

Nous évoquons dans cette section les problèmes posés pour la réalisation d'une bibliothèque d'interface en C. La représentation des données introduit une difficulté liée à la nature du polymorphisme que permet le langage ; elle doit être compatible avec l'utilisation de valeurs externes au langage et s'intégrer dans les systèmes de types respectifs (question traitée différemment pour un langage typé statiquement comme ML ou typé dynamiquement comme Scheme). Se posent également des problèmes de structures de contrôle et de gestion mémoire (particulièrement pour les langages munis de récupérateur automatique de mémoire).

Encapsulation et copie de valeurs. La manipulation d'une valeur structurée, et de ses sous-structures, d'un langage par un autre est la première étape de communication. Ces valeurs peuvent être partagées ou copiées d'un monde à l'autre. Dans le premier cas on passe par un entête qui encapsule la zone mémoire de la valeur; dans le deuxième cas on alloue une nouvelle zone mémoire pour contenir la conversion de la valeur d'un monde à l'autre. En règle générale les valeurs immédiates sont copiées et les valeurs structurées sont partagées.

Pour plus d'information, la problématique d'encapsulation de valeurs et de vérification de types est clairement exposée dans la thèse de Thierry Saura [72].

Gestion mémoire. Plus que l'encapsulation/partage de données, le point délicat provient de la gestion (allocation/libération) de mémoire. Les langages fonctionnels utilisent tous un mécanisme de GC qui garantit l'absence de pointeurs fantômes ou de zones mémoire inaccessibles. Un programme construit en C et dans un langage fonctionnel possède deux zones d'allocation dynamique. La première en C est gérée à la main alors que la deuxième est gérée automatiquement par la bibliothèque d'exécution du langage fonctionnel. Le risque est alors de perdre les propriétés de sûreté du GC. Les difficultés proviennent de la bonne constitution de l'ensemble des racines à conserver dans le tas des structures de données hétérogènes (contenant des valeurs des deux milieux), en particulier pour les structures circulaires.

Même si le langage fonctionnel est compilé vers du C, à la manière de Bigloo [74] ou CeML [18], il faut encore que le GC utilisé scrute la pile d'exécution C et traite bien les valeurs C à conserver, à la manière du GC généraliste de Boehm [7]. Ce GC a été conçu pour des programmes C ou des compilateurs utilisant C comme langage cible. Il a été éprouvé par de nombreuses implantations mais il est moins performant que des GC spécialisés comme celui d'Objective Caml. Cela est principalement dû à son algorithme `Mark&Sweep`, qui en plus de ne pas compacter le tas, a une complexité linéaire sur la taille totale du tas et non pas sur la taille des objets vivants du tas.

Fonctions. Le protocole d'appel des fonctions utilise des primitives de copie ou de partage de valeurs entre les deux mondes. Cela autorise les appels par valeur et par référence. Le passage par nom (macros) peut être simulé par une fermeture qui gèle l'expression à calculer. Les fermetures peuvent être appelées en C en utilisant le mécanisme général d'application de la bibliothèque d'exécution du langage fonctionnel.

L'appel à du code C à partir de ML varie selon le langage cible du compilateur. Par exemple CeML compile un dialecte ML vers du C de haut niveau ce qui facilite le mécanisme d'appel (voir 4.1.1.3). Pour Objective Caml le code engendré par le compilateur natif suit le protocole d'appel de C et autorise la communication dans les deux sens (à partir et vers C). Pour le compilateur de code-octet d'Objective Caml l'interprète de code-octet doit être embarqué dans l'application pour pouvoir exécuter des fonctions ML à partir de C.

Exceptions. De même que les différents appels de fonction passent d'un monde à l'autre, les récupérateurs d'exception s'imbriquent aussi. Bien souvent, les plates-formes d'exécution divergent dans l'implantation des récupérateurs d'exception. Le `setjmp/longjmp` de C n'est pas reconnu par le bytecode d'un `try/with` Objective Caml. Il est nécessaire de transformer la valeur de l'exception lors de la traversée d'un monde à l'autre, quitte à revenir à son état initial au retour dans son monde d'origine.

Multi-threading. Un dernier point délicat dans l'interfaçage avec C est la cohabitation de *threads* dont les contextes ne sont pas cohérents entre les plates-formes d'exécution. Le GC tient compte en général des piles d'exécution des différents threads comme ensemble de racines à conserver. La création de nouveaux *threads* en C non liés au GC peut provoquer une récupération intempestive de valeurs encore utilisées. Pour éviter cela Objective Caml utilise un *mutex* global dans les appels à du code C en mode *multithreadé*.

4.1.1.3 Utilisation des interfaces de C

L'utilisation des entêtes C permet d'automatiser la construction de valeurs et l'appel de fonctions C. Un choix est alors d'intégrer cet outil au langage fonctionnel ou bien d'utiliser un outil externe.

Langage d'interface intégré au langage. Le projet CeML [18] est un exemple d'extension du langage Caml permettant d'automatiser la construction de valeurs et l'appel de fonctions C. Le langage d'interface choisi est constitué d'un sous-ensemble des déclarations des fichiers interface C (`.h`). Pour cela une nouvelle directive `extern` a été introduite. Elle permet d'analyser une déclaration C (incluant les types, les variables globales et les macros) dans le but de construire de nouveaux types et fonctions Caml pour les manipuler. Ces nouveaux types sont considérés abstraits par le typeur Caml.

Une déclaration de structure C dans une primitive `extern` de CeML a pour effet du côté Caml de déclarer un type abstrait correspondant à la structure ainsi que des fonctions d'accès et de modification pour chacun de ses champs. Les fonctions engendrées sont monomorphes. Des fonctions analogues sont produites pour permettre l'accès à des tableaux ou des pointeurs. Les fonctions C sont prises en charge et il est possible de les appliquer partiellement en utilisant CeML.

L'intérêt de ce type d'intégration est au moins double :
- ne pas être dépendant des évolutions des spécifications des interfaces C,
- avoir l'information de typage de la partie Caml et de la partie C en même temps.

Outil d'encapsulation pour la communication. Une autre solution plutôt que d'intégrer la syntaxe des fichiers d'en-têtes C à Caml est d'utiliser un outil externe sachant lire de tels fichiers dans le but d'engendrer le code Caml d'encapsulation. L'intérêt est de ne pas modifier son langage préféré. La difficulté est de ne pas pouvoir limiter les entêtes à traiter comme lors de l'intégration précédente. Ces outils sont particulièrement utiles quand il faut traduire un ensemble de fichiers d'en-têtes interdépendants. SWIG (*Simple Wrapper and Interface Generator*) [3] en est un exemple dont il existe une version pour Objective Caml.

Ces outils sont aussi utilisés pour les extensions objet de C comme C++ et Objective C. C'est le cas pour C++ avec SWIG. La création d'un objet devient l'appel d'une fonction de création; la surcharge est remplacée par un mécanisme de nommage. L'appel d'une méthode est aussi traduit par un appel de fonction dont un argument est l'objet receveur. Néanmoins ces outils s'avèrent limités pour le traitement de communication dans les deux sens (*callbacks*).

Les systèmes de types peuvent poser des contradictions pour une génération automatique de code encapsulant. Une technique utilisée est de n'engendrer automatiquement que les parties compatibles et d'ignorer les autres, ce qui s'avère particulièrement intéressant pour les bibliothèques importantes. C'est ce qui a été réalisé avec les bibliothèques `Foundation` et `appKit` d'Objective C pour NextStep par Jérôme Vouillon et repris par Guy Cousineau pour Mac OS X.

Ces outils restent pratiques pour une communication simple entre C et un autre langage. Pour une collaboration plus riche il devient nécessaire de décrire plus précisément l'interaction souhaitée.

4.1.1.4 Les langages de définition d'interfaces (IDL)

L'utilisation de langages de définitions d'interface (IDL) permet de décrire les signatures des fonctions, procédures et méthodes qui seront appelées entre deux ou plusieurs langages. Le fait qu'un tel langage soit indépendant d'un langage de programmation permet un certain niveau de standardisation tout en évitant de compliquer chaque langage de programmation par l'intégration du langage d'interface. Souvent ils sont construits pour des besoins d'interopérabilité, y compris pour des applications réparties, comme pour DCE [91] de l'*Open Group*. Une grande famille d'IDL est orientée pour l'interfaçage des composants COM de Microsoft, comme par exemple H/Direct [38] et CamlIDL [55].

Néanmoins la création d'IDL dépend le plus souvent d'un besoin spécifique d'interfaçage d'un langage ou d'un mécanisme de composants liés à certaines caractéristiques de programmation. Par exemple l'IDL de CORBA [75] défini par l'*OMG* (*Open Management Group*) est partie intégrante de la spécification de *CORBA*. Cet IDL était orienté pour C++. Son évolution prend la marque de Java (passage par copie d'objets).

4.1.2 Quelques implantations pour Objective Caml

De nombreux efforts on été faits pour ouvrir Objective Caml à d'autres langages de programmation, par le biais de différentes interfaces s'inscrivant dans les tendances définies à la section 4.1.1.1.

4.1.2.1 Interfaces de bas niveau

Elles existent pour de nombreux langages autres que C. Citons notamment :

• Interface avec Java : **Camljava** [53] propose une interface similaire à JNI pour Objective Caml. Le principe est de communiquer à travers C, par l'interface native de Caml d'une part et JNI d'autre part. L'interface est faiblement typée (les références à des objets Java sont cachées derrière un type abstrait) et propose un jeu de primitives très semblable à JNI pour manipuler des objets Java (instanciation, appel de méthodes, accès aux champs). Un mécanisme de *callback* élémentaire (permettant l'appel de code Caml depuis Java) est disponible mais nécessite d'écrire du code souche Java à la main.

• Suivant les mêmes principes, il existe une interface avec Python **pycaml** [67], qui réplique l'interface Python/C dans un module Caml, ainsi qu'avec TCL/TK **CamlTK** [70].

• L'interface avec Perl **perl4caml** [49] propose une interface de bas-niveau permettant d'appeler du code Perl depuis Caml et certaines enveloppes de haut niveau autour de bibliothèques CPAN (*Comprehensive Perl Archive Network*). L'interface de bas niveau propose des fonctions de conversion de types Caml et trois types abstraits utilisés respectivement pour les valeurs scalaires, les tableaux et les valeurs de hachage de Perl, un mécanisme d'appel de méthodes Perl ainsi que l'évaluation d'une expression Perl contenue dans une chaîne de caractères (à la manière de la fonction `eval` de Perl, ce qui est bien peu typé).

4.1.2.2 Compilateurs d'interfaces

• **FFI** [39] prend en entrée des fichiers d'en-tête C et engendre un mélange de code C et de déclarations externes Caml. Les types de données C qui ne peuvent être directement sérialisés vers Caml en utilisant son interface C standard (comme les structures) sont stockés sous forme de chaînes de caractères afin d'être pris en charge par le GC. **Cigen** [80] est un projet similaire.

• **oDLL** [14] fabrique une DLL Windows à partir d'une bibliothèque Caml. L'utilitaire prend en entrée des fichiers Caml compilés (interfaces compilées `.cmi` et fichiers objets de bibliothèques `.cma`/`.cmxa`) et génère le code souche C nécessaire ainsi que les fichiers d'en-tête permettant d'utiliser la DLL dans du code C.

• Du même auteur, **CamlOLE** [13] permet l'utilisation de composants COM depuis Caml. Cela consiste en une bibliothèque de bas niveau permettant l'interfaçage et un utilitaire `OLEGen` qui engendre automatiquement les fichiers d'implantation et d'interface Caml encapsulant un composant COM à partir de sa bibliothèque de types OLE. On obtient de la sorte une interface sûre et bien typée masquant les détails bas-niveau des appels à COM.

• **Forklift** [46] est à mi-chemin entre la compilation d'interface et la compilation d'IDL. En effet il génère du code souche C et une interface Caml permettant l'utilisation de code C, en se basant sur les fichiers d'en-tête d'une part, et d'autre part sur des annotations exprimées dans une algèbre de souches et conservées dans un fichier séparé.

4.1.2.3 Compilateurs d'IDL

• La boîte à outils **SWIG** [3] permettant de connecter des programmes C et C++ à une grande variété de langages de programmation de haut niveau (le credo de SWIG est d'interfacer un langage de script avec des routines C/C++) dispose d'un générateur Objective Caml [2]. SWIG travaille sur des fichiers IDL qui correspondent à des en-têtes C munis d'indications supplémentaires guidant les sérialisations. SWIG définit un langage de *typemaps* permettant les correspondances de valeurs structurées les plus complexes. Il est également possible de gérer la transmission d'exceptions entre C++ et le langage de haut-niveau : il faut pour cela rédiger le détail des gestionnaires d'exceptions dans le fichier IDL.

• **camlORBit** [1] est une interface avec ORBit2 [66], l'ORB CORBA du projet GNOME. Un compilateur d'IDL traduit les types CORBA en types Caml : en plus des types de bases, les structures CORBA sont mises en correspondance avec les enregistrements Caml, les unions discriminées avec des variants et les énumérations avec des variants formés de constructeurs constants. Les interfaces CORBA sont représentées au moyen de types d'objets Caml. L'héritage d'interfaces et les exceptions CORBA sont pris en charge. Les fichiers générés sont liés à une bibliothèque de support mixte C/Caml.

• **SimpleSOAP** [48] est un client SOAP expérimental pour Caml. Il utilise des IDL décrivant les services Web au moyen d'une syntaxe basée sur les fichiers d'interface Caml et ajoutant des indications d'URI. Leur compilation fournit le code souche nécessaire à la sérialisation des arguments d'un appel de fonction distante, la requête HTTP au service et l'analyse syntaxique du résultat. Pour l'instant ce client ne traite qu'un sous-ensemble des valeurs Caml d'une part (essentiellement des enregistrements ou des listes d'enregistrements de types de base) et qu'un sous-ensemble des spécifications SOAP d'autre part.

• **OCamlIDL** [55] est à la fois un générateur de code souche C et une interface avec COM :

- Le code souche C est engendré à partir d'une IDL Microsoft (MIDL, *Microsoft's Interface Description Language* [58]). Cela ressemble à des fichiers d'en-têtes C avec annotations, plus une notion d'interface d'objets, sans héritage. OCamlIDL permet l'utilisation haut-niveau de code C depuis Caml et réalise une interface entre classes C++ et classes Objective Caml, dans les deux directions. Le modèle objet sous-jacent à cette communication est toutefois assez réduit : il se limite à un polymorphisme d'interfaces, sans notion d'héritage.

– Une bibliothèque de fonctions permet l'utilisation de composants COM en Caml et réciproquement l'encapsulation de code Caml derrière une interface COM.

• **OJacaré** [20] est un générateur de code qui facilite l'interopérabilité entre les langages Objective Caml et Java à travers leur modèle objet respectif et qui se base sur un IDL simple correspondant à un modèle objet à l'intersection de ceux des deux langages.

Pour les communications de Java vers Objective Caml est ajouté un mécanisme de rappel (*callback*). L'implantation repose sur les interfaces de bas niveau avec C de chaque langage (*JNI*, *Java Native Interface* [43] pour Java et `external` pour Objective Caml) et utilise une version étendue de la bibliothèque `camljava` [53]. OJacaré engendre toutes les classes encapsulantes nécessaires de manière conforme aux deux systèmes de types. Des vérifications de typage s'effectuent d'une part à la compilation en engendrant du code contenant des contraintes de type et d'autre part durant la phase d'élaboration, au chargement de l'application, par une inspection dynamique des classes Java chargées. Bien que l'IDL soit à l'intersection des deux modèles objet, OJacaré permet de combiner certaines caractéristiques des deux.

4.2 La plate-forme OCamIL / OJacaré.NET

4.2.1 Présentation de OJacaré.NET

OJacaré.NET est une adaptation de l'outil OJacaré [20] au monde .NET. Lors du développement de OJacaré de nombreuses difficultés sont venues de la gestion de deux environnements d'exécution différents (celui de Java et celui de Objective Caml), particulièrement en ce qui concerne la prise en charge des threads et de la récupération automatique de mémoire. L'utilisation de l'implantation OCamIL dans le cas de OJacaré.NET rend les choses beaucoup plus simples en raison de l'unicité de l'environnement d'exécution. De plus cela permet d'interfacer à Objective Caml la plupart des langages portés sur .NET puisque la communication se fera au travers du bytecode CIL.

4.2.1.1 Intersection des modèles objet et définition de l'IDL

Comparaison des modèles objet. Objective Caml intègre à son système de types une extension orientée objets basée sur des classes, pour laquelle les deux relations d'héritage et de sous-typage sont clairement distinctes [71, 21].

Une déclaration de classe définit :
– un nouveau type abréviation d'un type objet,
– une fonction de construction permettant d'instancier des objets de la classe.

Un type objet est caractérisé par le nom et le type de ses méthodes. Par exemple le type suivant peut être inféré pour les instances de classes possédant des méthodes `moveto` et `toString` :

```
< moveto : (int * int) -> unit; toString : unit -> string >
```

Les seules méthodes que l'on peut appeler sur un objet sont celles visibles dans son type. Le typage statique garantit alors que l'objet receveur possède bien une méthode du bon nom et du bon type. L'exemple 1 qui suit est correct si la classe `point` définit (ou hérite) une méthode `moveto` prenant une paire d'entiers comme argument. Les objets s'intègrent dans le modèle fonctionnel typé par les types objets ouverts (`<..>`). De même dans l'exemple 2, la fonction `f` acceptera comme argument n'importe quel objet dont le type contient une méthode `moveto` dont l'argument est une paire d'entiers.

Ex1 : appel de méthode	Ex2 : style fonctionnel et objet
let p = **new** point(1,1);; p#moveto(10,2);;	# **let** f o = o # moveto (10,20);; **val** f : < moveto : int * int –> 'a; .. > –> 'a

Les déclarations de classes acceptent l'héritage multiple et les classes paramétrées. Par contre la surcharge de méthode n'est pas acceptée. Du point de vue de l'exécution la liaison est toujours tardive (*late binding*).

La table suivante compare les caractéristiques principales du modèle objet du CTS avec celles de Objective Caml :

Caractéristiques	CTS	Objective Caml
Classes	$\surd$	$\surd$
Liaison tardive	$\surd$	$\surd$
Liaison précoce	$\surd$	1
Typage statique	$\surd$	$\surd$
Typage dynamique	$\surd$	2
Sous-typage	$\surd$	$\surd$
Héritage $\equiv$ sous-typage	oui	non
Surcharge	$\surd$	3
Héritage multiple	4	$\surd$
Classes paramétrées	$\surd$5	$\surd$
Paquetages/modules	6	6

Remarques :

1) Les méthodes statiques sont des fonctions globales d'un module Objective Caml ; on peut avoir des variables de classe.

2) Pas de *downcast* en Objective Caml, seulement dans l'extension coca-ml [19].

3) Pas de surcharge en Objective Caml mais le type de `self` peut apparaître dans le type d'une méthode qui pourra être redéfinie (*overriding*) dans une sous-classe.

4) Pas d'héritage multiple pour les classes CTS, seulement pour les interfaces.

5) Les Generics [50] introduits à partir de C# 2.0 permettent une forme de paramétricité.

6) Les modules simples d'Objective Caml correspondent aux parties publiques des espaces de noms CTS; la notion de modules paramétrés est inexistante dans le

CTS.

L'intersection des deux modèles objet correspond à un langage structuré en classes, dont l'appel de méthode est toujours en liaison tardive. Les relations d'héritage et de sous-typage sont confondues. Du point de vue des types, il n'y a pas de surcharge, de plus le type de l'instance ne peut pas apparaître dans le type d'une méthode. L'héritage est simple et il n'y a pas de classes paramétrées. Ce modèle inspire un langage IDL simple pour interfacer les classes CTS et Objective Caml.

Définition de l'IDL. La grammaire de l'IDL utilisée par OJacaré.NET est donnée sur la figure 4.1 en syntaxe EBNF (*Extended Backus-Naur Form*).

```
idl := ("package" annotations? qid ';' enum* def+)+

def := modifier* annotations? "class" id ("extends" qid)? ("implements" intfs)? '{' decls? '}'
     | annotations? "interface" id "implements" intfs '{' decls? '}'

decls := decl | decl ';' decls

decl := modifier* annotations? type id               (* champ *)
      | modifier* annotations? type id '(' args? ')' (* méthode *)
      | annotations? "<init>" '(' args? ')'          (* constructeur *)

args := annotations? type | annotations? type ',' args

type := "boolean" | "byte" | "short" | "int" | "long"  | "char" | "string"
      | "float" | "double" | "object" | type "[]" | "void" | qid

intfs := qid | qid ',' intfs

modifier := "static" | "abstract"

annotations := '[' ann_list ']'

ann_list := ann | ann ',' ann_list

ann := "assembly" qid | "name" id | "callback"

enum := modifier* annotations? "enum" id '[' enumitems ']'
      | modifier* annotations? "flags" id '[' enumitems ']'

enumitems := annotations? id ('=' n)? | annotations? id ('=' n)? ',' enumitems

id := ...  (* identificateurs *)

qid := ... (* identificateurs avec espace de nom *)

n := ...   (* entiers signés *)
```

FIG. 4.1 – *La grammaire de l'IDL en syntaxe EBNF*

La motivation principale de ce travail est la facilité d'emploi pour les programmeurs .NET et Caml. Un fichier IDL déclare une liste de paquetages : chacun d'entre eux correspond à un espace de noms lié à un assemblage donné et déclare un certain nombre d'énumérations (nous les aborderons à la section 4.2.1.3), de classes, de classes abstraites et d'interfaces. Ces dernières correspondent respectivement à une classe, une classe abstraite et une interface CTS d'une part, et à une classe et des classes abstraites Objective Caml d'autre part.

L'héritage suit celui du CTS : simple pour les classes mais multiple pour les interfaces. Il est possible de déclarer des champs, des méthodes et des constructeurs. Les annotations `name` permettent de définir des alias utilisables par Objective Caml (nous verrons que c'est particulièrement utile pour les constructeurs). L'annotation `callback` est expliquée à la section 4.2.1.2.

Les types pris en charge dans les signatures couvrent la plupart des types de base, les tableaux et les types référence du CTS (via une référence de type qualifiée).

Utilisation de OJacaré.NET. OJacaré.NET se présente comme un outil en ligne de commande prenant en entrée un fichier IDL `p.idl` et générant du code d'encapsulation Objective Caml (fichiers `p.mli` et `p.ml` : le code client Objective Caml pourra se servir des définitions engendrées dans le module `P`) et optionnellement du code C# additionnel (voir la section suivante au sujet des deux niveaux d'encapsulation proposés).

Dans la suite nous illustrerons l'utilisation de OJacaré.NET et de son IDL à travers des exemples de communication entre programmes Objective Caml et C#. Le langage C# est très proche de la machine virtuelle et peut rendre compte de toutes les problématiques que nous avons rencontrées dans l'interopération avec OJacaré.NET.

4.2.1.2 Sémantique de l'interfaçage

Considérons le cas où un ficher IDL est écrit afin d'utiliser dans un programme Objective Caml du code CIL existant. Un certain nombre de classes Objective Caml sont engendrées dans un module source après utilisation de OJacaré.NET. Du point de vue du programmeur un appel de Objective Caml vers du code CIL externe se fait alors par un appel de méthode, et le sens inverse par un appel à une méthode redéfinie en Objective Caml.

Les variables d'instance ou de classe CTS sont accessibles depuis Objective Caml par des méthodes engendrées automatiquement. Du point de vue de la sémantique des échanges de valeurs, seules les valeurs des types de base sont donc passées (ou retournées) par copie, dans tous les autres cas (objets, tableaux...) les valeurs sont transmises par référence, assurant ainsi le partage. Les exceptions sont propagées d'un monde à l'autre.

Enfin le code engendré à partir d'un fichier interface est sûr du point de vue du typage par une vérification des classes CTS au chargement du programme.

Le modèle de communication proposé entre Objective Caml et C# est stratifié en deux niveaux :

- le premier niveau permet un mécanisme d'encapsulation simple des objets C# dans les objets Objective Caml,
- le deuxième ajoute un mécanisme de callback autorisant la réimplantation (*overriding*) de méthodes C# en Objective Caml au moyen de la liaison tardive.

Nous décrivons ici le mécanisme d'interfaçage et l'utilisation de OJacaré.NET. Les détails d'implantation sont donnés à la section 4.2.2.

Encapsulation de premier niveau. En prenant pour point de départ la description de classes et d'interfaces dans un fichier IDL, OJacaré.NET génère du code d'enveloppe dans le langage cible (ici, Objective Caml) qui permet d'allouer des objets et d'appeler des méthodes sur les classes définies dans le langage externe (ici, C#) comme s'il s'agissait de classes natives.

Illustrons ce mécanisme sur un exemple simple : nous voulons utiliser en Caml deux classes C#, `Point` et `ColoredPoint` définies dans un assemblage `point`. La classe de « point » est à coordonnées entières dans le plan et sa classe fille « point coloré » ajoute une couleur représentée par une chaîne de caractères. Dans cet exemple la méthode `toString` de la classe `ColoredPoint` retourne la concaténation d'un appel à la méthode `toString` de `super` et d'un appel à la méthode `getColor` sur `this`.

Fichier `p.idl`

```
package [assembly point] mypack;

class Point {
  int x; int y;

  [name default_point] <init> ();
  [name point] <init> (int,int);
  void moveTo(int,int);
  string toString();
  void display();
  boolean equals(Point);
}
interface Colored {
  string getColor();
  void setColor(string);
}

class ColoredPoint extends Point
                       implements Colored {
  [name default_colored_point] <init> ();
  [name colored_point] <init> (int,int,string);

  [name equals_pc] boolean equals(ColoredPoint)
}
```

OJacaré.NET force l'utilisateur à gérer lui-même le problème de la surcharge (impossible en Caml) par l'attribution de noms différents, comme on peut le voir sur les constructeurs des classes ci-dessus (annotations `[name]`).

Un exemple d'utilisation du module P engendré en Caml est illustré dans une session de toplevel ci-dessous.

Session toplevel Objective Caml	
# **open** P;; # **let** p = **new** point 1 2;; val p : point = <obj> # **let** p2 = **new** default_point ();; val p2 : default_point = <obj> # **let** pc = **new** colored_point 3 4 "blue";; val pc : colored_point = <obj> # **let** pc2 = **new** default_colored_point ();; val pc2 : default_colored_point = <obj> # p#toString ();; - : string = "(1,2)"	# pc#toString ();; - : string = "(3,4):blue" # p#equals (pc :> cPoint);; - : bool = false # pc#moveTo 1 2;; - : unit = () # pc#equals p;; - : bool = true # pc#equals_pc pc2;; - : bool = false

L'opérateur de coercition de type :> permet de ne voir le type d'un objet que comme un sur-type.

Encapsulation complète : le mécanisme de callback. Poursuivons avec l'exemple précédent. Nous voulons redéfinir la méthode `getColor` en Objective Caml, et ainsi spécialiser la méthode `toString` grâce à la liaison tardive. Avec une encapsulation élémentaire, une instance C# de `ColoredPoint` n'a aucune connaissance de l'instance Objective Caml, comme on peut le voir dans l'exemple suivant, où on hérite d'une enveloppe de base.

```
# class colored_point_ml x y c =
  object
    inherit
        colored_point x y c as super
    method getColor () =
        "ML" ^ super#getColor ()
  end;;
class colored_point_ml :
            int -> int -> string -> cColoredPoint
# let wml_cp =
    new colored_point_ml 6 7 "green";;
val wml_cp : colored_point_ml = <obj>
# wml_cp#toString ();;
- : string = "(6,7):green"
```

Nous avons besoin d'un deuxième niveau de communication, déclenché par l'attribut `callback`[1] :

[callback] class ColoredPoint **extends** Point **implements** Colored { ... }

1. Nous verrons que l'attribut `callback` provoque la génération de classes additionnelles. Le prototype de OJacaré.NET fait en sorte que ce ne soit pas le comportement par défaut pour des raisons de performance et de complexité de l'édition de liens.

Avec cet attribut, la compilation du fichier `p.idl` génère un nouveau fichier en C# appelé `ColoredPointStub.cs` et ajoute des classes souches au fichier Objective Caml engendré. Comme illustré dans l'exemple suivant, hériter de la souche en Objective Caml donne cette fois-ci le comportement attendu.

```
# class mixed_colored_point x y c =
  object
    inherit
        callback_colored_point x y c as super
    method getColor () =
        "ML" ^ super#getColor ()
  end;;
class mixed_colored_point :
          int -> int -> string -> cColoredPoint
# let mixed_cp =
    new mixed_colored_point 8 9 "red";;
val mixed_cp : mixed_colored_point = <obj>
# mixed_cp#toString ();;
- : string = "(8,9):MLred"
```

La figure 4.2 montre la différence d'organisation des classes engendrées avec ou sans l'attribut `callback`. Nous le détaillons dans la section 4.2.2.2.

4.2.1.3 Constructions complémentaires.

Types valeurs structurés. Il est possible d'utiliser des types valeurs définis par l'utilisateur : les classes déclarées dans l'IDL peuvent aussi bien désigner un type référence ou un type valeur (ce qui bien sûr impose automatiquement des restrictions sur les déclarations valides ; par exemple on ne peut pas hériter d'un type valeur). En pratique ce sont les formes encapsulées (donc des types références) qui sont manipulées du côté Caml : elles sont donc incarnées par des classes Objective Caml de manière homogène avec les types références.

Énumérations. Les énumérations *à la* C# sont partiellement prises en compte par l'IDL, au moyen du mot-clé `enum`. Seules les énumérations dont le type sous-jacent est un entier sont interfaçables (ce qui constitue la grande majorité des cas).

Les énumérations sont restituées en Objective Caml au moyen d'un type variant dont tous les constructeurs sont constants. Cela permet une très bonne intégration syntaxique, puisque les différentes alternatives déclarées dans une énumération peuvent être envoyées sur des constructeurs de même nom (ce qui peut toujours être ajusté au moyen d'attributs `name` de l'IDL). Par exemple :

<table>
<tr><th>Fichier IDL</th><th>Session toplevel</th></tr>
<tr><td><pre>
enum Color {
  Red,
  Green,
  Blue
}

class EnumColoredPoint {
  int x; int y;
  Color c;
  [name epoint] <init> (int,int,Color);
  string toString()
}
</pre></td><td><pre>
let col = Blue;;
- : enum_Color = Blue
let p = new epoint 1 2 col;;
val p : epoint = <obj>
p#toString ();;
- : string = "(1,2):blue"
p#get_c ();;
- : enum_Color = Blue
</pre></td></tr>
</table>

Il est possible de spécifier une ou plusieurs des valeurs sous-jacentes de l'énumération (par des entiers positifs ou négatifs), les valeurs étant prises croissantes en dehors de ces indications, la première valant 0 par défaut.

OJacaré.NET traite également les énumérations de type *flag* (marquées en C# par l'attribut `[FlagsAttribute]`), qui permettent de combiner les constantes de l'énumération par un « ou » logique afin de gérer des champs de bits. On utilise dans l'IDL le mot-clé `flags`. Dans ce cas les valeurs Caml associées sont des listes de constructeurs du type énuméré (c'est-à-dire que pour l'exemple ci-dessous, l'équivalent de l'expression C# « `One | Five` » en Caml sera « `[One;Five]` ».

<table>
<tr><th>Énumération avec valeurs assignées</th><th>Énumération de type flag</th></tr>
<tr><td><pre>
enum NoZero {
  MinusTwo = -2,
  MinusOne,
  One = 1,
  Two
}
</pre></td><td><pre>
flags Power2 {
  None,
  One,
  Two,
  Three = 0x08,
  Four = 0x10,
  Five = 0x20
}
</pre></td></tr>
</table>

Propriétés. Il n'y a pas de construction dédiée aux propriétés CTS dans l'IDL. Cependant chaque propriété CTS est implantée sur la plate-forme .NET au moyen de méthodes de lecture et d'écriture, qui à leur tour sont accessibles à l'outil OJacaré.NET.

Par exemple pour utiliser une propriété `P` de type entier en lecture et écriture, il suffit de déclarer dans l'IDL les méthodes sous-jacentes `int get_P();` et `void set_P(int);`.

4.2.2 Implantation

4.2.2.1 Interface de bas niveau

L'architecture d'interopérabilité de OCamIL est construite sur une base extrêmement simple d'appels externes à du code CIL depuis les programmes Objective Caml.

Encapsulation des valeurs et interface d'appels. Nous avons remplacé la prise en charge par le compilateur Objective Caml d'appels externes à des fonctions de bibliothèques écrites en C (mot-clé `external`) par la possibilité d'appeler des méthodes statiques CIL. Bien sûr ce mécanisme est indépendant du langage qui était à la source de ce code CIL, cela peut être C# ou autre chose.

Nous avons largement employé ce procédé dans l'adaptation à OCamIL de la bibliothèque standard de Objective Caml : le code C a été remplacé par des appels à des méthodes de `core_camil.dll` et même parfois par des appels directs à la BCL. Voici par exemple un extrait du module `Sys` de la bibliothèque standard :

```
external il_getenv: string -> string =
   "string" "System.Environment" "GetEnvironmentVariable" "string"
;;
let getenv var = match il_getenv var with
    | "" -> raise Not_found
    | s -> s
;;
```

La version OCamIL est faite d'un zeste de code Objective Caml enveloppant un appel direct à la méthode statique `GetEnvironmentVariable` de l'espace de noms `System.Environment` de la BCL.

Une déclaration `external` comporte deux versants, qui permettent de comprendre ses capacités et ses limitations :

- La spécification de la méthode à appeler par sa signature (sous forme de chaînes de caractères dont la syntaxe est analysée par le compilateur OCamIL).
- La signature Caml nécessaire au typeur afin de gérer la fonction comme une fonction Caml.

Toute incohérence entre les deux formes de signature peut résulter en des erreurs dynamiques de types lors de l'exécution des appels par la machine .NET, si bien qu'il faut considérer cette interface de bas-niveau comme non-sûre.

Le langage utilisé pour la spécification de la signature CTS est limité. Les seuls types valeurs pris en charge sont les types de base, mais aucun type utilisateur. Les types références (y compris utilisateur) sont tous reconnus (mot-clé `class` suivi de la référence de type). Les tableaux sont traités mais pas les types délégués. Les références de types acceptent espaces de noms et assemblages (par exemple `"class [mscorlib]System.Text.StringBuilder"`). Les lacunes de ce langage sont claires : il n'y a aucun moyen d'instancier une classe CTS quelconque et les appels de méthodes CIL sont restreints aux seules méthodes statiques.

Gestion des exceptions. Le passage d'exceptions entre les mondes Caml et pur CIL se fait de manière transparente. Tout d'abord les exceptions Caml sont représentées au moyen des exceptions de la plate-forme .NET, ce qui les rend exploitables dans tout code CIL externe. Réciproquement, nous voulons aussi pouvoir gérer des exceptions quelconques de la plate-forme dans le code Caml. Pour cela il suffit de modifier légèrement la compilation des gestionnaires d'exceptions (voir la section 3.1.3.3) de façon à ce qu'ils s'activent sur le type partagé par toutes exceptions `System.Exception`. Un test de type permet de discriminer les exceptions Caml des autres : ces dernières sont alors encapsulées dans une exception Caml particulière (`CLIException`) afin de pouvoir poursuivre le traitement de manière homogène.

```
CODE SAUVANT L'ÉTAT DE LA PILE DANS DES VARIABLES LOCALES
try {
   CODE COMPILÉ POUR t1
   stloc VAL //sauver le résultat de t1 dans une variable locale VAL
   leave EXIT
}
catch System.Exception {
   //la valeur d'exception est sur la pile
   dup
   isinst     CamIL.Exception
   brfalse    EMBED
   castclass  CamIL.Exception
   br         CAMILEXC
EMBED
   call       class CamIL.Exception
                        CamIL.Exception::embedCLI(class System.Exception)
CAMILEXC
   ldfld      object[] CamIL.Exception::v1
   stloc.2
   CODE COMPILÉ POUR t2 //remarque : e est sur la pile à l'entrée du bloc
   stloc VAL //sauver le résultat de t2 dans une variable locale VAL
   leave EXIT
}
EXIT:
CODE RÉCUPÉRANT L'ÉTAT DES VARIABLES LOCALES
ldloc VAL //récupérer le résultat de l'évaluation du try..with
```

Le compilateur OCamIL étend la bibliothèque standard de Caml en définissant l'exception `CLIException` :

```
exception CLIException of string * cli_exception
```

Lors de l'encapsulation de l'exception CLI dans cette exception Caml (par l'appel de la méthode `embedCLI` ci-dessus), l'objet contenant l'exception complète est stocké dans le deuxième paramètre sous forme d'un type abstrait `cli_exception` qui peut-être analysé à l'aide d'une boîte à outil (basée sur `System.Reflection`) fournie dans la bibliothèque standard de OCamIL, alors que le nom du type exception est stocké sous forme d'une chaîne de caractères dans le premier argument. Cela donne un moyen commode pour filtrer les exceptions à rattraper, par exemple :

```
try ...
with
   | Invalid_argument x -> ...
   | CLIException ("System.StackOverflowException",e) -> ...
```

En cas d'échec du filtrage, le compilateur OCamIL utilise l'instruction `rethrow` qui permet de relancer l'exception : cela fonctionne de manière homogène avec les exceptions CLI générales ou spécifiques à OCamIL car c'est l'exception initialement capturée (et pas une version éventuellement encapsulée) qui est relancée.

Divers. La récupération automatique de mémoire et le multi-threading ne présentent pas de problème particulier puisqu'ils reposent sur les mécanismes fournis par la machine .NET et sont uniformes entre un composant OCamIL et les autres.

4.2.2.2 La génération de code souche

Bibliothèque de support. OJacaré.NET utilise une bibliothèque de bas niveau servant de support au code souche engendré. Cette bibliothèque expose au code Caml un certain nombre de services de base proposé par l'API `Reflection` : chargement de types, appels de méthodes et de constructeurs, accès aux champs, vérification de signatures etc...

Elle est constituée de deux parties :

- La première partie, initialement codée en C#, a été désassemblée en CIL et intégrée au runtime OCamIL : celle-ci expose les méthodes utiles de `Reflection` sous une forme utilisable par la FFI de OCamIL (voir la sous-section 4.2.2.1), n'utilisant que des méthodes statiques par exemple.
- La deuxième partie est formée d'un module Caml `Jacare` ajouté à la bibliothèque standard qui fait le lien à la première au moyen de déclaration `external`.

En principe le module `Jacare` n'est pas appelé par le code utilisateur, mais seulement de manière interne au code Caml engendré par OJacaré.NET. Si on considère une communication de haut niveau telle qu'un appel de méthode, l'exécution passe successivement par le code souche engendré, le module `Jacare`, puis la bibliothèque correspondante dans l'environnement d'exécution `core_camil.dll`, l'API `Reflection`, et enfin le code cible de l'appel dans un composant quelconque.

On peut noter que la bibliothèque de bas niveau étant intégrée à l'environnement OCamIL, l'utilisation de OJacaré.NET dans les programmes OCamIL ne complique pas l'édition de liens.

Génération du code souche. La compilation d'un fichier `.idl` construit les classes nécessaires pour une communication sûre, du point de vue typage, gestion mémoire et gestion d'exceptions au dessus de la bibliothèque de support de OJacaré.NET.

Nous présentons ici la situation où OJacaré.NET est utilisé pour exploiter des classes CTS depuis Objective Caml. La principale difficulté est que les classes CTS décrites dans l'IDL et les classes Caml engendrées pour leur correspondre ne sont pas représentées de manière identique sur la plate-forme .NET. La génération de code par OJacaré.NET repose sur l'architecture suivante :

– les objets CTS sont effectivement transmis au langage Caml, mais sous la forme de valeurs masquées par un type abstrait `Jacare.obj`,
– les classes Objective Caml engendrées enveloppent ces objets, répliquent en Caml les interactions possibles pour chaque classe de l'IDL et s'occupent de sérialiser les messages entre les deux mondes,
– la distinction de Objective Caml entre types objets et classes permet de pallier à l'absence de surcharge dans un cadre typé.

Notons que pour faciliter la représentation de la hiérarchie de classes CTS, nous introduisons une classe Objective Caml racine `CtsHierarchy.top` dont héritent les classes engendrées.

Le tableau suivant décrit les types et les classes engendrés en Objective Caml et C# par la compilation d'une déclaration dans l'IDL selon la présence ou non de l'attribut `callback`.

<table>
<tr><td>pour une classe</td><td>pour une interface</td></tr>
<tr><td>- 1 type objet t
- 1 classe encapsulante W de type t
- 1 à n classes (C_i), sous-classes de W (1 par constructeur)
- 1 fonction <code>instanceof</code> pour ce type
- 1 fonction de cast pour ce type</td><td>- 1 type objet t
- 1 classe encapsulante W de type t
- 1 fonction <code>instanceof</code> pour ce type
- 1 fonction de cast pour ce type</td></tr>
<tr><td colspan="2">en ajoutant l'attribut <code>callback</code></td></tr>
<tr><td>- 1 classe souche (stub)
- 1 à n classes abstraites (1 par constructeur) dont toutes les méthodes sont concrètes</td><td>- 1 classe abstraite dont toutes les méthodes sont abstraites</td></tr>
<tr><td>- 1 sous-classe C#</td><td>- 1 classe C# implantant l'interface</td></tr>
</table>

Les types références du CTS sont vus en Objective Caml comme des types objets t : ceux-ci expriment le contrat d'interface implanté par une classe et permettent de réaliser une relation de sous-typage conforme à celle du CTS.

La classe enveloppe W se construit sur une instance `Jacare.obj` de la classe CTS. Elle garde une référence sur cette instance et implante toutes les méthodes déclarées dans le type objet t au moyen de sérialisations vers et depuis l'instance enveloppée. Cependant la classe W ne peut être instanciée en Caml qu'à l'aide de ses classes filles C_i, une par constructeur déclaré dans l'IDL. Le constructeur de C_i sérialise ses arguments en direction du constructeur correspondant du module externe, et récupère l'instance nouvellement créée sous forme d'une valeur abstraite de type `Jacare.obj`, qui permet alors l'instanciation de W. Toutes les classes C_i et W partagent le même type objet t. Les différentes classes C_i contournent l'absence de surcharge en Objective Caml.

Pour chaque type t équivalent à une type référence, une fonction `instance_of_`t permet de tester si un objet est bien de type t et une fonction de *cast* permet de réaliser effectivement le transtypage (qui consiste simplement à construire une enveloppe de type objet Caml différent sur une même instance d'objet CTS de type `Jacare.obj`).

Dans l'exemple du `ColoredPoint` de la section 4.2.1.2 on engendrera donc un type objet `cColoredPoint` contenant toutes les méthodes décrites dans l'IDL et deux classes « utilisateur » `colored_point` et `default_colored_point` héritant d'une même classe encapsulante `wrapper_cColoredPoint` (les trois classes ont le type `cColoredPoint`).

Voici le code des types objets Caml engendrés :

```
type _cts_cPoint = Jacare.obj
type _cts_cColored = Jacare.obj
type _cts_cColoredPoint = Jacare.obj

class type cPoint =
  object
    inherit CtsHierarchy.top
    method _get_cts_cPoint : _cts_cPoint
    method set_x : int -> unit
    method get_x : unit -> int
    method set_y : int -> unit
    method get_y : unit -> int
    method moveTo : int -> int -> unit
    method toString : unit -> string
    method equals : cPoint -> bool
  end
and cColored =
  object
    inherit CtsHierarchy.top
    method _get_cts_cColored : _cts_cColored
    method getColor : unit -> string
    method setColor : string -> unit
  end
and cColoredPoint =
  object
    inherit cPoint
    inherit cColored
    method _get_cts_cColoredPoint : _cts_cColoredPoint
    method equals_pc : cColoredPoint -> bool
  end
```

Les types abstraits abritant les instances d'objets CTS sont des alias à `Jacare.obj`. Chaque type objet déclare une méthode permettant de récupérer cet objet CTS encapsulé. L'héritage multiple de Objective Caml est utilisé pour refléter simultanément les relations d'héritage de classe et d'implantation d'interface du CTS.

Passons au code des classes encapsulantes :

```
class _wrapper_cPoint =
  let clazz = Jacare.find_class "point" "mypack.Point" in
  let __fid_x = Jacare.get_fieldID clazz "x" Jacare.Tint in
  let __fid_y = Jacare.get_fieldID clazz "y" Jacare.Tint in
  let __mid_moveTo =
    Jacare.get_methodID clazz "moveTo" ([| Jacare.Tint; Jacare.Tint |], Jacare.Tvoid) in
  let __mid_toString =
    Jacare.get_methodID clazz "toString" ([| |], Jacare.Tstring) in
  let __mid_equals = Jacare.get_methodID clazz "equals" ([| Jacare.Tclazz clazz |], Jacare.Tbool) in

  fun (cts_ref : _cts_cPoint) ->
    let _ = if Jacare.is_null cts_ref then raise (CtsHierarchy.Null_object "mypack.Point") in
    object (self)
      method set_x _p = Jacare.set_int_field cts_ref __fid_x _p
      method get_x () = Jacare.get_int_field cts_ref __fid_x
      method set_y _p = Jacare.set_int_field cts_ref __fid_y _p
      method get_y () = Jacare.get_int_field cts_ref __fid_y
      method moveTo _p0 _p1 =
        Jacare.call_void_method cts_ref __mid_moveTo [| Jacare.Int _p0; Jacare.Int _p1 |]
      method toString () = Jacare.call_string_method cts_ref __mid_toString [| |]
      method equals (_p0 : cPoint) =
        let _p0 = _p0#_get_cts_cPoint in
        Jacare.call_boolean_method cts_ref __mid_equals [| Jacare.Obj _p0 |]
      method _get_cts_cPoint = cts_ref
      inherit CtsHierarchy.top cts_ref
    end

and _wrapper_cColored =
  let clazz = Jacare.find_class "point" "mypack.Colored" in
  let __mid_getColor = Jacare.get_methodID clazz "getColor" ([| |], Jacare.Tstring) in
  let __mid_setColor = Jacare.get_methodID clazz "setColor" ([| Jacare.Tstring |], Jacare.Tvoid) in

  fun (cts_ref : _cts_cColored) ->
    let _ = if Jacare.is_null cts_ref then raise (CtsHierarchy.Null_object "mypack.Colored") in
    object (self)
      method getColor () = Jacare.call_string_method cts_ref __mid_getColor [| |]
      method setColor _p0 = Jacare.call_void_method cts_ref __mid_setColor [| Jacare.Tstring _p0 |]
      method _get_cts_cColored = cts_ref
      inherit CtsHierarchy.top cts_ref
    end

and _wrapper_cColoredPoint =
  let clazz = Jacare.find_class "point" "mypack.ColoredPoint" in
  if not (Jacare.is_assignable_from clazz (Jacare.find_class "point" "mypack.Point"))
     then failwith "Wrong super class in IDL: mypack.ColoredPoint does not extend mypack.Point.";
  if not (Jacare.is_assignable_from clazz (Jacare.find_class "point" "mypack.Colored"))
     then failwith "Wrong implemented interface in IDL:
                   mypack.ColoredPoint does not implement mypack.Colored.";
  let __mid_equals_pc =
    Jacare.get_methodID clazz "equals" ([| Jacare.Tclazz clazz |], Jacare.Tbool) in

  fun (cts_ref : _cts_cColoredPoint) ->
    let _ = if Jacare.is_null cts_ref then raise (CtsHierarchy.Null_object "mypack.ColoredPoint") in
    object (self)
      method equals_pc (_p0 : cColoredPoint) =
        let _p0 = _p0#_get_cts_cColoredPoint in
        Jacare.call_boolean_method cts_ref __mid_equals_pc [| Jacare.Obj _p0 |]
      method _get_cts_cColoredPoint = cts_ref
      inherit _wrapper_cColored cts_ref
      inherit _wrapper_cPoint cts_ref
    end
```

Chaque capsule contient un prélude définissant des variables de classes : celles-ci exploitent la bibliothèque de support de OJacaré.NET pour identifier chaque classe, méthode ou champ déclaré dans l'IDL (grâce à l'API `Reflection`). Ces opérations permettent de déceler à l'initialisation du programme Caml toute inadéquation entre les déclarations de l'IDL et les classes effectivement accessible à l'exécution : chacune d'entre elles lève une exception en cas d'échec.

Les appels à `Reflection` via la bibliothèque de support de OJacaré.NET nécessitent de décrire les types CTS reconnus par l'IDL : le module `Jacare` déclare un type variant :

```
type type_cts = Tbool | Tint | ...
              | Tobject | Tclazz of clazz | Tarray of type_cts
```

Nous verrons également dans la suite (lors des définitions de méthodes) l'utilisation d'un type variant encapsulant des valeurs Caml correspondantes :

```
type value_cts = Boolean of bool | Int of int | ... | Object of obj
```

Le reste du code des capsules met en évidence la construction (prenant en argument une référence d'objet CTS derrière un type `Jacare.obj`), et la définition des méthodes consistant à sérialiser tous les appels vers et depuis cette instance encapsulée. Les champs sont transformés en accesseurs (méthodes `get` et `set`).

L'instanciation d'une classe à l'aide de la capsule est faite de la manière suivante dans le code engendré :

```
let _init_point =
  let clazz = Jacare.find_class "point" "mypack.Point" in
  let id = Jacare.get_constructorID clazz [| Jacare.Tint; Jacare.Tint |] in
  fun _p0 _p1 -> Jacare.call_constructor id [| Jacare.Int _p0; Jacare.Int _p1 |]

let _init_default_point =
  let clazz = Jacare.find_class "point" "mypack.Point" in
  let id = Jacare.get_constructorID clazz [| |] in
  fun () -> Jacare.call_constructor id [| |]

let _init_colored_point =
  let clazz = Jacare.find_class "point" "mypack.ColoredPoint" in
  let id = Jacare.get_constructorID clazz [| Jacare.Tint; Jacare.Tint; Jacare.Tstring |] in
  fun _p0 _p1 _p2 ->
    Jacare.call_constructor id [| Jacare.Int _p0; Jacare.Int _p1; Jacare.String _p2 |]

let _init_default_colored_point =
  let clazz = Jacare.find_class "point" "mypack.ColoredPoint" in
  let id = Jacare.get_constructorID clazz [| |] in
  fun () -> Jacare.call_constructor id [| |]

class point _p0 _p1 =
  let cts_obj = _init_point _p0 _p1 in
  object (self) inherit _wrapper_cPoint cts_obj end

class default_point () =
  let cts_obj = _init_default_point () in
  object (self) inherit _wrapper_cPoint cts_obj end

class colored_point _p0 _p1 _p2 =
  let cts_obj = _init_colored_point _p0 _p1 _p2 in
  object (self) inherit _wrapper_cColoredPoint cts_obj end

class default_colored_point () =
  let cts_obj = _init_default_colored_point () in
  object (self) inherit _wrapper_cColoredPoint cts_obj end
```

Il y a autant de classes que de constructeurs surchargés. Chacune hérite de la capsule correspondant à la classe CTS de l'IDL. La construction d'une instance CTS est réalisée au moyen de `Reflection` et la valeur de type `Jacare.obj` qui en résulte est passé au constructeur de la capsule.

En présence de l'attribut `callback` deux nouvelles classes « souche » sont engendrées, l'une en Objective Caml et l'autre en C#, comme illustré sur la figure 4.2.

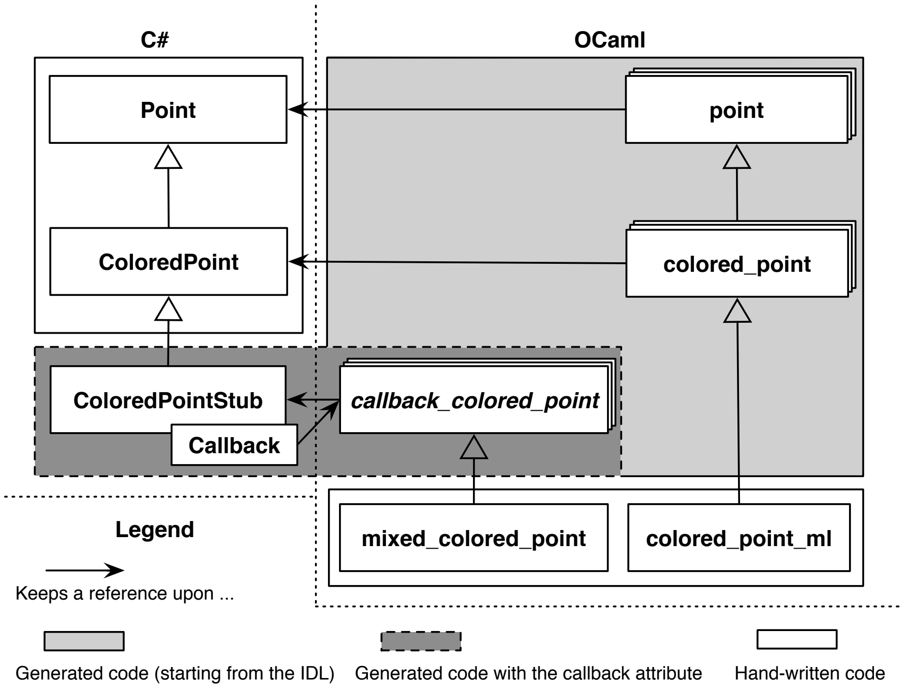


FIG. 4.2 – *Relations entre classes*

Les deux souches en C# et Objective Caml possèdent des références croisées l'une sur l'autre. Dans l'exemple de la figure 4.2, la souche C# `ColoredPointStub` hérite de la classe CTS `ColoredPoint` marquée par l'attribut *callback* dans l'IDL et redéfinit les méthodes de son ancêtre comme des appels vers la souche Objective Caml `callback_colored_point`. Inversement, la souche Objective Caml définit chaque méthode comme un appel non-virtuel aux méthodes de `ColoredPoint`, passant outre la souche `ColoredPointStub` afin d'éviter les cycles.

L'utilisateur peut alors définir ses propres comportements en héritant d'une classe « utilisateur » (héritière directe de la classe souche Objective Caml) en sachant que le comportement par défaut de ces classes « utilisateur » est de propager l'appel vers la classe ancêtre CTS.

Voici le début du code additionnel engendré par OJacaré.NET en présence de l'attribut *callback* devant la classe `ColoredPoint` :

```
class type virtual _stub_cColoredPoint =
  object
    inherit CtsHierarchy.top
    method _get_cts_cPoint : _cts_cPoint
    method _get_cts_cColored : _cts_cColored
    method _get_cts_cColoredPoint : _cts_cColoredPoint
```

```
    method _stub_equals_pc : Jacare.obj -> bool
    method _stub_setColor : Jacare.obj -> unit
    method _stub_getColor : Jacare.obj
    method _stub_equals : Jacare.obj -> bool
    method _stub_toString : Jacare.obj
    method _stub_moveTo : int -> int -> unit
    method equals_pc : cColoredPoint -> bool
    method virtual setColor : string -> unit
    method virtual getColor : unit -> string
    method equals : cPoint -> bool
    method toString : unit -> string
    method moveTo : int -> int -> unit
    method set_y : int -> unit
    method get_y : unit -> int
    method set_x : int -> unit
    method get_x : unit -> int
  end
```

Le type objet précédent déclare les membres de la classe `callback_colored_point` en cumulant les membres définis par les interfaces et le long des relations d'héritage qui sont prises en compte par sa contrepartie `ColoredPoint`.

De plus ces membres sont doublés : les méthodes dont le nom commence par `_stub_` sont appelées par la classe C# `ColoredPointStub` et leur fonction est d'effectuer des sérialisations mineures sur les objets Caml avant d'appeler les véritables méthodes de la classe.

Le code d'implantation de `callback_colored_point` suit. On peut voir par exemple que dans le cas de la méthode `equals_pc`, la méthode `_stub_equals_pc` est non triviale : elle s'occupe de transformer un objet de type `ColoredPoint` passé par le code C#, caché derrière un type Caml abstrait, en une instance de classe Caml (cette opération est plus simple à effectuer dans le code Caml que depuis C#, d'où l'utilité des méthodes `_stub_`).

```
...
and virtual callback_colored_point =
  let cbclazz = Jacare.find_class "callback_point" "mypack.ColoredPointStub" in
  let clazz = Jacare.find_class "point" "mypack.ColoredPoint" in
  let _clazz_cPoint = Jacare.find_class "point" "mypack.Point" in
  let __midInit =
    Jacare.get_methodID cbclazz "__setCamlObj" ([| Jacare.Tarray Jacare.Tobject |], Jacare.Tvoid) in
  let __mid_equals_pc =
    Jacare.get_methodID clazz "equals" ([| Jacare.Tclazz clazz |], Jacare.Tbool) in
  let __mid_equals =
    Jacare.get_methodID clazz "equals" ([| Jacare.Tclazz _clazz_cPoint |], Jacare.Tbool) in
  let __mid_toString = Jacare.get_methodID clazz "toString" ([| |], Jacare.Tstring) in
  let __mid_moveTo =
    Jacare.get_methodID clazz "moveTo" ([| Jacare.Tint; Jacare.Tint |], Jacare.Tvoid) in
  let __fid_y = Jacare.get_fieldID clazz "y" Jacare.Tint in
  let __fid_x = Jacare.get_fieldID clazz "x" Jacare.Tint in

  fun (cts_ref : _cts_cColoredPoint) ->
    let _ = if Jacare.is_null cts_ref then raise (CtsHierarchy.Null_object "mypack.ColoredPoint") in
    object (self)
      method _stub_equals_pc (_p0 : _cts_cColoredPoint) =
        let _p0 : cColoredPoint = new _wrapper_cColoredPoint _p0 in self#equals_pc _p0
      method _stub_setColor _p0 = self#setColor _p0
      method _stub_getColor = self#getColor ()
      method _stub_equals (_p0 : _cts_cPoint) =
        let _p0 : cPoint = new _wrapper_cPoint _p0 in self#equals _p0
      method _stub_toString = self#toString ()
      method _stub_moveTo _p0 _p1 = self#moveTo _p0 _p1
      method equals_pc (_p0 : cColoredPoint) =
        let _p0 = _p0#_get_cts_cColoredPoint in
        Jacare.call_nonvirtual_boolean_method cts_ref __mid_equals_pc [| Jacare.Obj _p0 |]
```

```
    method virtual setColor : string -> unit
    method virtual getColor : unit -> string
    method equals (_p0 : cPoint) =
      let _p0 = _p0#_get_cts_cPoint in
      Jacare.call_nonvirtual_boolean_method cts_ref __mid_equals [| Jacare.Obj _p0 |]
    method toString () = Jacare.call_nonvirtual_string_method cts_ref __mid_toString [| |]
    method moveTo _p0 _p1 =
      Jacare.call_nonvirtual_void_method cts_ref __mid_moveTo [| Jacare.Int _p0; Jacare.Int _p1 |]
    method set_y _p = Jacare.set_int_field cts_ref __fid_y _p
    method get_y () = Jacare.get_int_field cts_ref __fid_y
    method set_x _p = Jacare.set_int_field cts_ref __fid_x _p
    method get_x () = Jacare.get_int_field cts_ref __fid_x
    method _get_cts_cColoredPoint = cts_ref
    initializer Jacare.call_void_method cts_ref __midInit [|Jacare.Obj (Obj.magic self)|]
    method _get_cts_cPoint = cts_ref
    method _get_cts_cColored = cts_ref
    inherit CtsHierarchy.top cts_ref
  end
```

Dans le code précédent, `"callback_point"` (ligne 2) fait référence à l'assemblage dans lequel est compilé le code C# supplémentaire engendré par OJacaré.NET. On peut noter l'existence d'une méthode `__setCamlObj` (ligne 6) déclarée par la classe C# et qui sert à passer une référence de l'objet Caml dérivant de la classe `callback_colored_point` à l'objet C#. Cette opération est effectuée au niveau de l'`initializer` de l'objet Caml : nous donnons les détails de la construction d'un objet un peu plus loin.

Comme expliqué plus haut, on peut vérifier que les méthodes de la classe souche sont implantées par des appels non virtuels aux méthodes de `ColoredPoint`.

Le code suivant sert à construire une instance de `callback_colored_point`.

```
let _init__stub_colored_point =
  let clazz = Jacare.find_class "callback_point" "mypack.ColoredPointStub" in
  let id = Jacare.get_constructorID clazz [| Jacare.Tint; Jacare.Tint; Jacare.Tstring |] in
  fun _p0 _p1 _p2 ->
     Jacare.call_constructor id [| Jacare.Int _p0; Jacare.Int _p1; Jacare.String _p2 |]

let _init__stub_default_colored_point =
  let clazz = Jacare.find_class "callback_point" "mypack.ColoredPointStub" in
  let id = Jacare.get_constructorID clazz [| |] in fun () -> Jacare.call_constructor id [| |]

class virtual _stub_colored_point _p0 _p1 _p2 =
  let cts_obj = _init__stub_colored_point _p0 _p1 _p2 in
  object (self) inherit callback_colored_point cts_obj end

class virtual _stub_default_colored_point () =
  let cts_obj = _init__stub_default_colored_point () in
  object (self) inherit callback_colored_point cts_obj end
```

Pour finir, voici le code C# engendré : celui-ci doit être compilé dans une DLL indépendante définissant un assemblage `callback_point`.

```
namespace mypack {
class ColoredPointStub : mypack.ColoredPoint {
  private object[] cb;
  public void __setCamlObj(object[] obj) {this.cb = obj;}
  public ColoredPointStub(int _p0, int _p1, string _p2) : base(_p0, _p1, _p2) {}
  public ColoredPointStub() : base() {}

  private static int __mid_moveTo =
    (int)(CamlinternalOO.closures._new_method.exec("_stub_moveTo"));
  public override void moveTo(int _p0, int _p1) {
    object[] args = {(object)(_p0), (object)(_p1)};
```

```
    CamIL.Closure closure = (CamIL.Closure) Camlinternal00.closures._send.exec(this.cb,__mid_moveTo);
    for(int i = 0 ; i < args.Length - 1 ; i++) closure = (CamIL.Closure) closure.apply(args[i]);
    object res = closure.apply(args[args.Length-1]);
    return ;
  }

  private static int __mid_toString =
    (int)(Camlinternal00.closures._new_method.exec("_stub_toString"));
  public override string toString() {
    object res = Camlinternal00.closures._send.exec(this.cb,__mid_toString);
    return (string)res;
  }

  private static int __mid_equals =
    (int)(Camlinternal00.closures._new_method.exec("_stub_equals"));
  public override bool equals(mypack.Point _p0) {
    object[] args = {_p0};
    CamIL.Closure closure = (CamIL.Closure) Camlinternal00.closures._send.exec(this.cb,__mid_equals);
    for(int i = 0 ; i < args.Length - 1 ; i++) closure = (CamIL.Closure) closure.apply(args[i]);
    object res = closure.apply(args[args.Length-1]);
    return (((int)(res))!=0);
  }

  /* codes de getColor et setColor omis dans cette présentation */

  private static int __mid_equals_pc =
    (int)(Camlinternal00.closures._new_method.exec("_stub_equals_pc"));
  public override bool equals(mypack.ColoredPoint _p0) {
    object[] args = {_p0};
    CamIL.Closure closure = (CamIL.Closure) Camlinternal00.closures._send.exec(this.cb,__mid_equals_pc);
    for(int i = 0 ; i < args.Length - 1 ; i++) closure = (CamIL.Closure) closure.apply(args[i]);
    object res = closure.apply(args[args.Length-1]);
    return (((int)(res))!=0);
  }
}
}
```

La création d'un objet de classe `mixed_colored_point` (suivant la notation de la figure 4.2) s'effectue en 3 étapes :

- allocation d'un objet CTS `pcs` de classe `ColoredPointStub` : ceci est réalisé dans les fonctions `_init__stub_default_colored_point` (respectivement `_init__stub_colored_point`) utilisées à la construction des deux classes respectives `_stub_default_colored_point` et `_stub_colored_point`,
- initialisation de l'objet Objective Caml `pcm` avec la référence sur `pcs` : c'est pris en compte par le constructeur de `callback_colored_point` qui prend cette référence en paramètre,
- initialisation de l'objet CTS `pcs` : appel au constructeur de `ColoredPoint` (effectué à la construction de l'objet `pcs` dont la classe `ColoredPointStub` dérive de `ColoredPoint`) et initialisation de l'objet `cb` avec une référence sur `pcm` : ceci est réalisé par l'`initializer` de la classe `callback_colored_point` qui effectue un appel de la méthode `__setCamlObj` de `pcs` (au moment de l'appel d'un initialiseur en Objective Caml, l'objet est déjà construit et on peut utiliser `self`).

Les appels de *callback* du code externe vers les classes Objective Caml se font moyennant l'exploitation des fonctions du module `CamlInternalOO` de la bibliothèque standard de Objective Caml à la manière d'une API. Ces fonctions sont accessibles dans l'assemblage `core_camil` auquel le code C# souche peut naturellement se lier, comme tout composant .NET.

La figure 4.3 donne le détail de la communication entre code Caml et code externe lors d'un appel de méthode (la pile d'appels se lit de bas en haut). Entre crochets sont figurés les appels non-virtuels de Caml à C#.

Pile .NET

C#	`pc.getColor()`	pc
Caml	`pcm#getColor()`	[pc]
Caml	`pcm#getColor()`	pcm
C#	`pc.getColor()`	pcs
C#	`pc.toString()`	pc
Caml	`pcm#toString()`	[pc]
C#	`pcs.toString()`	pcs
C#	`pc.display()`	pc
Caml	`pcm#display()`	[pc]
Langage		**Classe de déf.↑**

FIG. 4.3 – *Détail d'un appel à display sur un* `mixed_colored_point`

Notons une limitation du mécanisme de *callback*: celui-ci ne peut pas être mis en jeu lors de l'appel d'un constructeur. Supposons par exemple que le code du constructeur de `ColoredPoint`, prenant en arguments les coordonnées du point et sa couleur, utilise la méthode `setColor` pour assigner la couleur. Il n'est pas possible avec notre mécanisme de *callback* de spécialiser le code de `setColor` dans une classe Objective Caml afin de produire un effet dans le constructeur de `ColoredPoint`. Lorsqu'on instancie la souche `ColoredPointStub`, celle-ci appelle le constructeur de `ColoredPoint`, ce qui amène en vertu de la liaison tardive à aller chercher une spécialisation de `setColor`, y compris du côté Caml. Or la souche `ColoredPointStub` n'en est qu'à la première phase de sa construction et ne possède pas encore de référence sur un objet Caml, ce qui empêche de résoudre la liaison dans la classe spécialisée en Caml.

Autres constructions. Les types valeurs sont gérés exactement de la même manière que les types référence (sauf bien-sûr qu'ils sont plus simples : n'ayant pas de méthode d'instance, le principe du *callback* ne s'applique pas à eux).

Les méthodes de l'API `Reflection` travaillent toujours sur des versions boxées des valeurs, c'est pourquoi d'une part il n'a pas été nécessaire d'implanter une sérialisation spécifique pour les types valeurs, et d'autre part ceux-ci sont traités comme des types références du côté Caml (c'est-à-dire obligatoirement sous forme boxée).

En ce qui concerne les énumérations, il s'est surtout agi de mettre en place les sérialisations. Comme cas particulier de type valeur, une classe Objective Caml interface chaque type énumération et donne accès en lecture au champ `value__` (celui-ci est présent dans tous les types énumérations du CTS et donne l'entier sous-jacent à chaque valeur d'une énumération). Cette classe n'est pas exposée au code Caml client mais est utilisée en interne pour sérialiser du CTS vers le type variant représentant l'énumération en Caml : pour cela il suffit de récupérer la valeur entière sous-jacente et de la transtyper au moyen de `Obj.magic` (puisqu'un type variant dont tous les constructeurs sont constants est représenté par un entier dans le runtime OCamIL). Si l'IDL assigne des valeurs non régulières aux membres de l'énumération, on utilise une table globale pour associer à chaque valeur son index.

Dans l'autre sens, se pose le problème de l'absence de constructeur de types énumérations. Les valeurs représentant les alternatives possible d'une énumération sont données par le CTS comme des champs statiques de la classe associée. Le code engendré par OJacaré.NET instancie une variable globale qui contient un tableau de ces valeurs (sous forme boxée) obtenues par des appels à `Reflection`. Pour sérialiser un constructeur du variant il suffit de le transtyper en entier (encore une fois par `Obj.magic`) et de l'utiliser comme index d'accès à l'élément correspondant du tableau.

4.2.2.3 Sûreté

Le typage statique de Objective Caml garantie la sûreté de l'exécution, mais que se passe-t-il lorsqu'il est interfacé à du code externe ? On peut distinguer deux types d'erreurs :

- Les erreurs à l'exécution survenant dans du code CIL (des erreurs de transtypages dynamiques par exemple) qui ne résultent pas de l'interopération elle-même mais de problèmes dans le code externe.
- Des incohérences entre l'IDL et l'implantation. Par exemple, des composants CIL décrits dans l'IDL peuvent ne pas être disponibles lors de l'exécution, ou bien incorrectement décrits.

La première catégorie d'erreurs lève des exceptions : elles peuvent être considérées comme des erreurs d'exécution « normales ». Elles peuvent être rattrapées par le composant externe ou sinon par le code Objective Caml lui-même (voir 4.2.2.1).

La deuxième catégorie d'erreurs met directement en cause l'interopération. Nous choisissons de détecter ces erreurs le plus tôt possible. Les programmes Objective Caml qui font un usage incorrect de composants externes sont détectés à la compilation au moyen du typage statique mais celui-ci part de l'hypothèse que les types décrits dans l'IDL sont corrects, ce qui peut n'être attesté qu'au moment de l'exécution au moyen de la bibliothèque `Reflection`. Le code engendré par OJacaré.NET réalise des tests au lancement du programme par une allocation statique d'une instance de chaque objet représentant dans l'API `Reflection` une référence aux types, champs, constructeurs et méthodes déclarés dans l'IDL. En cas d'incohérence entre les composants résolus par le chargeur de classes de .NET et les fichiers IDL, une exception est lancée : celle-ci peut être rattrapée par le programme Caml ou autrement provoque l'interruption immédiate de l'exécution du programme et avertit l'utilisateur.

4.2.3 Expressivité et portée

L'exemple introductif de la section 4.2.1 ne mettait en jeu qu'une forme simple de communication. Dans cette section nous essayons d'aller un peu plus loin. Nous donnons d'abord un résultat positif sur l'expressivité du mélange entre les deux modèles objets. Puis nous appliquons la technologie OJacaré.NET sur un exemple concret qui nécessite une communication plus évoluée. Il sert de point de départ à une discussion sur les véritables limites de OJacaré.NET. Nos exemples utiliserons C#, sans restriction de généralité.

4.2.3.1 Combinaison des deux modèles objet

OJacaré.NET permet de profiter simultanément de certains aspects des deux modèles objets. Illustrons ces nouvelles possibilités en montrant un cas d'héritage multiple de classes C# en Objective Caml, et un exemple de vérification dynamique de type (*downcast*) en Objective Caml.

Héritage multiple de classes C#. L'exemple suivant est tiré de [21]. Nous définissons deux hiérarchies de classes en C# : les objets graphiques et les objets géométriques. Chaque hiérarchie possède une classe `Rectangle`.

Fichier `rect.idl`

```
package mypack;

class Point {
 [name point] <init> (int, int);
}
class GraphRectangle {
 [name graph_rect] <init>(Point, Point);
 string toString();
}
class GeomRectangle {
 [name geom_rect] <init>(Point, Point);
 double compute_area();
}
```

Le programme Objective Caml suivant définit une classe qui hérite des deux classes C#.

Programme Objective Caml

```
open Rect;;

class geom_graph_rect p1 p2 =
object
  inherit geom_rect p1 p2 as super_geo
  inherit graph_rect p1 p2 as super_graph
end;;

let p1 = new point 10 10;;
let p2 = new point 20 20;;
let ggr = new geom_graph_rect p1 p2;;
Printf.printf "area=%g\n" (ggr#compute_area ());;
Printf.printf "toString=%s\n" (ggr#toString ());;
```

Downcast d'objets C# en Objective Caml. En dehors de l'extension cocaml [19], Objective Caml ne permet aucune opération de typage dynamique sur les objets, ce qui s'avère pourtant nécessaire pour l'interopération avec C#, au moins pour les objets provenant d'un calcul du côté de C#. L'exemple ci-dessous montre le mélange de style fonctionnel et objet d'Objective Caml y compris pour des objets C# encapsulés. On y construit une liste `l` d'objets `cPoint`, alors même que ceux-ci sont des points colorés. Pour chaque hiérarchie de classes C# décrite dans un fichier IDL, OJacaré.NET génère une hiérarchie de classes Objective Caml, dont la classe racine est `CtsHierarchy.top`. OJacaré.NET génère également des fonctions de coercitions de types depuis `CtsHierarchy.top` vers chaque type Objective Caml correspondant à une classe C#. Ces fonctions lèvent une exception en cas d'incompatibilité de types.

```
let l = [(mixed_cp :> cPoint); (wml_cp :> cPoint)];;
val l : cPoint list = <obj>
let lc = List.map (fun x -> cColoredPoint_of_top (x :> top)) l;;
val l : cColoredPoint list = <obj>
```

4.2.3.2 Pertinence du modèle de communication

Complément : Objective Caml depuis les autres langages. Il faut insister sur la dissymétrie entre les composants Objective Caml et les autres, en raison de leurs modèles de classes. Les objets Objective Caml ne sont pas directement compilés vers des objets CTS et ils disposent de leur propre mécanisme d'héritage et de liaison retardée. Les appels depuis Objective Caml à un composant .NET quelconque utilisent les mécanismes d'introspection proposés par .NET mais dans l'autre sens les appels doivent prendre en compte les spécificités de l'implantation Objective Caml.

Dans ce qui précède nous avons présenté les principes d'un interfaçage Objective Caml-CTS axé sur Objective Caml, client de bibliothèques .NET et où les callbacks implantent les appels dans l'autre direction.

Nous avons étudié (mais pas implanté) la même approche pour la réciproque, c'est-à-dire de prendre en charge l'appel direct de méthodes Objective Caml depuis C# ou tout autre langage .NET : dans ce cas les appels directs reposent sur le module `CamlInternalOO` et les *callback* sur la bibliothèque `Jacare` et l'API de `Reflection` (il est toujours nécessaire d'implanter les deux niveaux de communication).

Il est possible de modifier l'outil OJacaré.NET pour qu'il prenne en charge cette deuxième configuration, prenant en argument un fichier IDL semblable à ce qui précède (la syntaxe est la même sauf que certaines constructions, comme les énumérations, ne sont pas disponibles).

Étant donné que Objective Caml n'a pas de mécanisme d'introspection des types, il faut adapter les mécanismes de sûreté. Même dans le cadre de l'encapsulation simple, nous générons du code Objective Caml qui va confronter statiquement l'IDL à l'implantation Objective Caml et aussi la vérifier dynamiquement contre le code externe au démarrage du programme.

Un exemple réel. Nous illustrons l'interopérabilité dans l'exemple suivant : étendre un programme Objective Caml avec une interface graphique écrite en C#. Nous partons de l'entrée ayant gagné la concours de programmation ICFP'2000 [84], un lanceur de rayons (*raytracer*) implanté en Objective Caml. Situons le problème :

- La classe Objective Caml `Render` définit une méthode `compute`. Cette méthode prend une chaîne de caractères en argument (le nom d'un fichier qui représente la scène 3D à dessiner) et une classe `Display` sur laquelle faire le rendu de pixels (au moyen d'appels à une méthode appelée `drawPixel`).
- L'interface graphique est une classe `Display` qui hérite de (ou bien conserve une référence sur une instance de) `System.Windows.Forms.Form`, le composant graphique racine de l'API de fenêtrage .NET. La classe `Display` est munie d'une méthode `drawPixel`, ce qui la rend compatible avec l'entrée de la méthode `compute` implantée en Caml. Une scène 3D est choisie avec une fenêtre de sélection de fichiers.

L'exécution effectue un va-et-vient entre les deux composants. Cela peut être implanté avec OJacaré.NET en utilisant la liaison tardive inter-langage. Deux solutions sont possibles (voir la figure 4.4).

Classe abstraite `Render`. La classe C# `Display` est paramétrée par une classe abstraite `Render` qui définit une méthode `compute` (son constructeur attend en argument une instance de cette classe). Le programme Objective Caml fournit une implantation de la classe `Render`. La méthode `Main` est du côté Objective Caml : elle passe une instance de `Render` au constructeur de `Display` (démarrant l'interface graphique). Quand l'utilisateur choisit un fichier de scène 3D, `compute` est appelée en lui passant la référence courante de l'interface graphique `Display` sur laquelle la méthode `drawPixel` peut être appelée depuis le code Caml pour chaque pixel calculé.

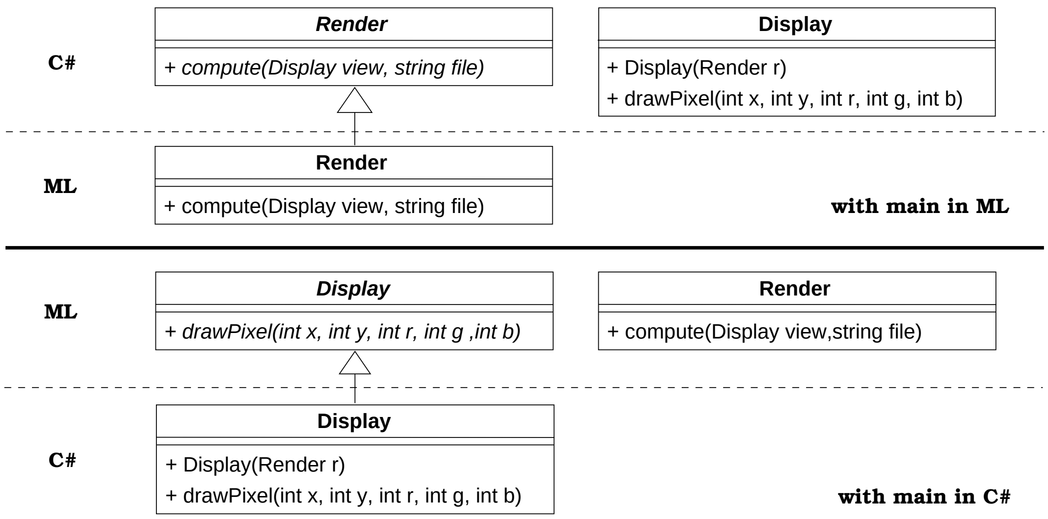


FIG. 4.4 – *Deux architectures pour l'interface graphique du lanceur de rayons*

Dans ce cas de figure un fichier IDL est utilisé pour faire connaître les classes `Display` et `Render` au code Caml : la première sert à typer l'argument de `compute` et à permettre les appels de `drawPixel` alors que la seconde est utilisée comme base de la classe Caml `Render` dérivée.

Classe abstraite `Display`. Dans cette autre architecture, Objective Caml définit une classe abstraite `Display` utilisée en entrée de la méthode `compute` et le code C# se charge de fournir une implantation à cette classe. Cette classe spécialisée garde une référence sur un objet chargé d'implanter l'interface graphique et qui dérive de la classe `System.Windows.Forms.Form` (elle ne peut pas en dériver car l'héritage multiple n'est pas possible en C#). La méthode `Main` est du côté C# : elle construit une instance de `Display` et lorsqu'un fichier de description de scène est sélectionné, le programme C# construit un objet `Render` en Objective Caml puis appelle `compute` sur le nom de fichier et `self` (le composant recevant le dessin).

Ici le fichier IDL déclare les deux classes Objective Caml utiles au code C# : la classe abstraite `Display` et la classe `Render` dont le programme doit appeler la méthode `compute`. L'outil OJacaré.NET est donc utilisé de manière opposée dans cette deuxième solution.

La figure 4.5 montre l'affichage du lanceur de rayons.

Perfectionnement du mélange. OJacaré.NET permet l'utilisation de composants d'un langage dans l'autre, dans les deux sens. Cependant cela n'est pas suffisant pour une synthèse parfaite des deux mondes.

Prenons l'exemple du lanceur de rayons. Si nous n'avions qu'un seul monde, il serait possible d'utiliser les fichiers IDL pour déclarer la méthode `drawPixel` de la classe C# `Display` et la méthode `compute` de la classe Objective Caml `Render`, les rendant accessible simultanément aux deux composants sans utiliser « l'astuce » de

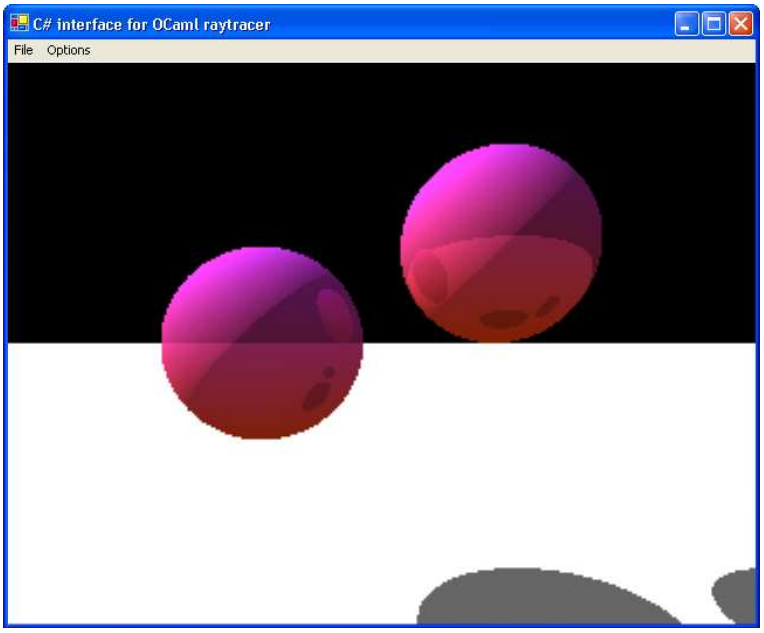


FIG. 4.5 – *Lancer de rayons Objective Caml sous .NET*

la redéfinition de classes abstraites. Malheureusement, les fichiers IDL ne peuvent être utilisés que pour exposer des classes d'un langage vers l'autre, à la fois.

On ne peut pas simuler un seul monde avec deux fichiers IDL, un pour décrire la classe C# `Display`, et un autre pour la classe Objective Caml `Render` car ces deux classes sont mutuellement récursives. Le problème est que la méthode `compute` attend une instance de `Display`, si bien que la deuxième IDL nécessite de décrire l'enveloppe Objective Caml de `Display` engendré à partir de la première IDL. Cela mène à des erreurs de typage lors de la compilation C# parce qu'il n'y a aucune relation d'héritage (de sous-typage en réalité) entre la classe `Display` de départ et la classe `Display` doublement encapsulée.

Une solution consisterait à ne pas orienter la génération de code par OJacaré.NET pour une cible ou une autre et voir un IDL comme la description de classes CTS devant interopérer, sans dissymétrie. C'est difficile à réaliser dans la mesure où les classes Objective Caml sont complètement différentes des classes du CTS, les objets Caml n'étant pas des objets de l'environnement d'exécution.

4.2.3.3 Génération automatique de fichiers IDL

Quand le nombre de classes CTS et de modules Objective Caml est important il peut sembler fastidieux de devoir écrire manuellement les différents fichiers IDL correspondants. Il est possible d'écrire un outil qui engendre automatiquement ces fichiers à la lecture des fichiers d'interfaces de Objective Caml et de l'inspection des types contenus dans un assemblage .NET au moyen de l'API `Reflection`.

Il y a plusieurs limites à cette approche :

- Dans leur état actuel, les fichiers IDL de OJacaré.NET manquent de compositionalité : il n'y a pas de directive `import`. Celle-ci ne pose pas de difficulté dans les cas simples, mais nous n'avons pas apporté de solution au problème des classes mutuellement récursives déclarées dans des IDL séparés.
- L'absence de surcharge nécessite des renommages, et il est difficile d'implanter des conventions de renommage automatique qui soient satisfaisantes sans intervention manuelle.
- L'IDL permet de ne déclarer que les classes utiles à une application donnée. Déclarer toutes les classes possibles entraîne des dégradations de performances, notamment en raison des mécanismes de sûreté qui inspectent les types de l'IDL au démarrage. Par exemple, la BCL compte environ 3000 classes publiques que l'on ne veut pas (peut pas...) soumettre au compilateur OCamIL sans discernement.

Pour ces raisons, nous avons conservé une approche manuelle dans notre prototype.

4.2.4 Travaux connexes

Les interfaces externes des langages fonctionnels sont passées de liens de bas niveau avec C et C++ pour Lisp [30], ML ou [37] à des IDL pour interfacer les composants COM pour Haskell [38, 56] et Objective Caml [55]. Toutes ces interfaces doivent gérer l'interaction des différents gestionnaires de mémoire, d'exceptions et de *threads*, le plus souvent en passant par C.

Les choix techniques d'intégration des environnements d'exécution peuvent différer. Plus généralement, l'approche au problème de l'interopérabilité varie d'une approche à l'autre.

4.2.4.1 Intégration des environnements d'exécution

L'approche de OJacaré (la version originale pour Java), consistant à faire cohabiter deux environnements d'exécution en parallèle, a également été utilisée dans d'autres projets. Ainsi l'interpréteur Haskell Husg98 for .NET [36] permet la manipulation de classes .NET. L'implantation est basée sur une mécanique semblable à Objective Caml / OJacaré. Au niveau du langage source, cela permet une communication sommaire avec la plate-forme .NET sur laquelle on construit une communication complète passant par des constructions haut-niveau du langage, ce qui nécessite la génération automatique de code par un outil dédié. Au niveau de l'exécution, on obtient deux machines (l'interprète et le CLR) tournant simultanément. Le projet Dot-Scheme [63] implante une interface d'appels externes vers la plate-forme .NET pour PLT Scheme. L'exécution est prise en charge là aussi par deux environnements. La manipulation de classes .NET se fait de manière assez transparente dans le langage, à partir d'une implantation qui repose sur les possibilités d'introspection du CLR.

La tendance actuelle est de changer d'assembleur portable pour produire directement du code-octet soit pour la machine Java comme MLj [4], soit pour .NET avec dans ce cas un nombre important de travaux, dont SML.NET [6] et MoscowML for .NET [51] pour SML, et F# [83] et OCamIL pour Objective Caml. Citons aussi Bigloo [11] qui propose entre autres des compilateur pour Java et .NET. L'intérêt est de n'avoir qu'une plate-forme d'exécution.

4.2.4.2 L'interopérabilité dans le cadre des langages fonctionnels sur .NET

F# et OCamIL illustrent bien deux points de vue différents sur l'interopérabilité. La conception de F# met l'accent sur l'interopérabilité elle-même : le but est de pouvoir manipuler dans un langage fonctionnel/impératif similaire à CamlLight le modèle objet proposé par la plate-forme .NET. Le résultat est un nouveau dialecte de Caml, dont le modèle objet est celui de .NET, ce qui est très différent du modèle objet introduit par Objective Caml. L'avantage est la manipulation directe des types CTS dans le langage même, sans aucun outil ou artifice supplémentaire. Cela procure un réel confort de programmation (on peut avoir l'impression d'avoir C# dans Caml) et rend l'implantation la plus directe possible, ce qui garantit de meilleures performances. En revanche, le modèle objet utilisé ne s'intègre pas aussi bien dans le paradigme fonctionnel que le modèle de Objective Caml. Il est régulièrement nécessaire d'aider l'inférence de types à l'aide d'annotations de types CTS, et le polymorphisme paramétrique et de rangée doit céder la place à un polymorphisme d'interfaces lorsque des appels de méthodes .NET sont mis en jeu.

Au contraire, OCamIL met l'accent sur le langage et prend le parti de compiler Objective Caml sur la plate-forme .NET sans l'altérer : en particulier le modèle objet de Objective Caml est conservé. Il n'a été rajouté aucune construction provenant de l'architecture cible et l'interopérabilité est gérée à travers les constructions du langage original. Cela a deux conséquences principales :

- la différence des deux modèles objets interdit une compilation directe des objets Caml vers le CTS : ce qui est gagné en expressivité se perd en efficacité,
- l'inadéquation précédente rend nécessaire la génération de classes souches, qui est automatisée par l'outil OJacaré.NET à partir de fichiers IDL.

Pour résumer, F# se positionne davantage du côté du programmeur .NET en lui proposant d'entrer dans le monde des langages fonctionnels alors que OCamIL se place du point de vue du programmeur Objective Caml qui veut bénéficier de l'environnement .NET sans rien sacrifier de son langage.

MLj et SML.NET réunissent les deux approches en proposant l'essentiel de SML sur les plates-formes Java et .NET, tout en enrichissant le langage à l'aide du modèle objet de ces plates-formes, à la manière de F# (mais il est vrai qu'en l'absence de traits objets dans le langage de départ, il n'y a pas de même arbitrage à faire que pour Objective Caml). MoscowML for .NET, quant à lui ne permet pour l'instant que l'appel de méthodes statiques.

Du côté de Scheme, le compilateur Bigloo permet de compiler vers la JVM ou le CLR. Comme pour Dot-Scheme, les traits .NET sont joliment incorporés au langage Scheme, par le biais de fonctions et de macros spéciales. Le langage Scheme se prête assez bien au jeu de l'interopérabilité : en effet sa syntaxe se prête naturellement aux extensions et l'absence de typage statique permet d'accommoder plus facilement les traits d'autres langages. Le typage dynamique est davantage dans l'esprit des plates-formes Java et .NET, qui proposent de nombreux services d'introspection.

Bien qu'Eiffel ne soit pas un langage fonctionnel, sa version .NET [77] rencontre des difficultés de compilation et d'interopérabilité qui semblent proches de celles d'Objective Caml. En effet les deux modèles objets possèdent un héritage multiple, des classes paramétrées sans avoir de surcharge. Cependant leurs techniques de compilation diffèrent fortement. Comme indiqué précédemment, les classes Objective Caml ne sont pas représentées par des classes équivalentes .NET, ce qui nécessite pour les besoins de l'interopérabilité de passer par un IDL pour encapsuler la représentation des objets de chaque langage. De son côté, Eiffel exploite les classes du CTS : chaque classe Eiffel est représentée d'une part par une interface (l'héritage multiple de Eiffel trouve écho dans l'héritage multiple des interfaces CTS) et d'autre part par une classe d'implantation privée. Cela permet d'utiliser simplement les classes Eiffel dans un autre langage .NET. Cependant pour être capable d'instancier des classes non différées (selon la terminologie Eiffel), le compilateur engendre des classes comportant des méthodes de création des classes d'implantation correspondantes.

Il est bien plus difficile d'exploiter directement les classes CTS dans la compilation de Objective Caml (même avec des aménagements comme ceux choisis par Eiffel) en raison du polymorphisme de rangées, des méthodes binaires et de la distinction entre les relations de sous-typage et d'héritage.

4.2.4.3 Enrichissement des langages

Plusieurs travaux ont permis par le passé d'enrichir le modèle objet de Java. Odersky et Wadler introduisent du polymorphisme paramétrique à Java dans leur système [61]. Ils ajoutaient ainsi à Java un style de programmation proche d'Objective Caml. L'évolution de Pizza vers Generic Java [10] reprend ce nouveau style en introduisant un polymorphisme borné. Cette volonté de généricité se retrouve bien entendu du côté de C# : la version 2.0 de .NET introduit des *Generics* [50, 92], et le langage intermédiaire ILX [82] se veut plus général que CIL pour faciliter la compilation de langages à la ML.

Du côté des langages fonctionnels la définition d'extensions objets comme Objective Caml autorise grâce à la liaison tardive de nouvelles structurations logicielles [9]. Néanmoins le besoin d'une véritable hiérarchie de classes est réel et une ouverture du dogme du typage statique [19] devient nécessaire pour la réalisation de certains modèles de conception (*Design Patterns*).

À la différence de MLj et SML.NET qui introduisent respectivement le modèle objet de Java et de C# dans des dialectes ML, les outils OJacaré et OJacaré.NET conservent les particularités de l'extension objet d'Objective Caml. Le résultat nous semble plus enrichissant en fusionnant aux polymorphismes paramétrique et de rangées d'Objective Caml et le typage dynamique de Java/C#.

Nous verrons dans le chapitre 5 des applications du mélange entre Objective Caml et .NET et effectuerons une analyses des performances au chapitre 6.

Chapitre 5

Applications et outils

Ce chapitre illustre des applications possibles de OCamIL grâce à ses capacités d'interopération avec d'autres composants .NET. Certaines ont recours à OJacaré.NET, et nous montrons un exemple d'utilisation de ce dernier dans le compilateur bootstrappé. Une application intéressante de OCamIL dans le domaine de la sérialisation sûre est donnée en fin de chapitre.

Sommaire

5.1 Écrire des composants OCamIL

Une grande motivation de notre travail vient des possibilités d'interopération entre composants .NET dont certains sont écrits en Objective Caml. Dans la suite nous décrivons différents moyens de faire interagir des composants OCamIL avec le monde .NET en nous appuyant sur des exemples concrets.

5.1.1 Interfaçage de composants OCamIL

5.1.1.1 Changer le contexte d'exécution des programmes OCamIL

Le compilateur d'IDL OJacaré.NET fournit un moyen riche de développer une application multi-langages faisant intervenir Objective Caml. Il existe des moyens plus directs pour faire communiquer des programmes Caml avec un environnement .NET sans pour autant revoir leur conception initiale.

Un programme Caml standard n'ayant pas été conçu initialement pour l'interopération dispose de plusieurs entrées-sorties de base :

- le système de fichiers,
- l'entrée standard, la sortie standard et la sortie d'erreurs,
- une fenêtre graphique (pour les applications se servant de la bibliothèque `Graphics` de la distribution Caml).

Nous proposons différents moyens de détourner ou de surcharger ces entrées-sorties afin d'établir des liens entre un programme Caml et des composants .NET externes.

Détourner `stdin`, `stdout` et `stderr`. Les types `in_channel` et `out_channel` définis par Caml pour l'entrée et les sorties standard (mais aussi pour les descripteurs de fichiers en lecture et écriture, voir la section suivante) masquent les types CLI `BinaryReader` et `BinaryWriter` de l'espace de noms `System.IO` de la BCL.

Ces classes peuvent être instanciées à partir d'un objet `System.IO.Stream` quelconque ce qui ouvre la porte à la personnalisation des flux d'entrées-sorties. Pour cela la bibliothèque d'exécution de OCamIL expose des méthodes permettant de substituer aux flux standards des objets `Stream` utilisateur :

- `void set_stdin(Stream s)`
- `void set_stdout(Stream s)`
- `void set_stderr(Stream s)`

Le programme OCamIL manipule ces flux de manière transparente alors même qu'ils sont surchargés.

En guise d'illustration nous utilisons un petit programme d'Alain Frisch qui recherche la solution à des Sudoku. La grille est soumise sous un format texte à l'entrée standard et la solution s'il elle existe est écrite sur la sortie standard (voir figure 5.1).

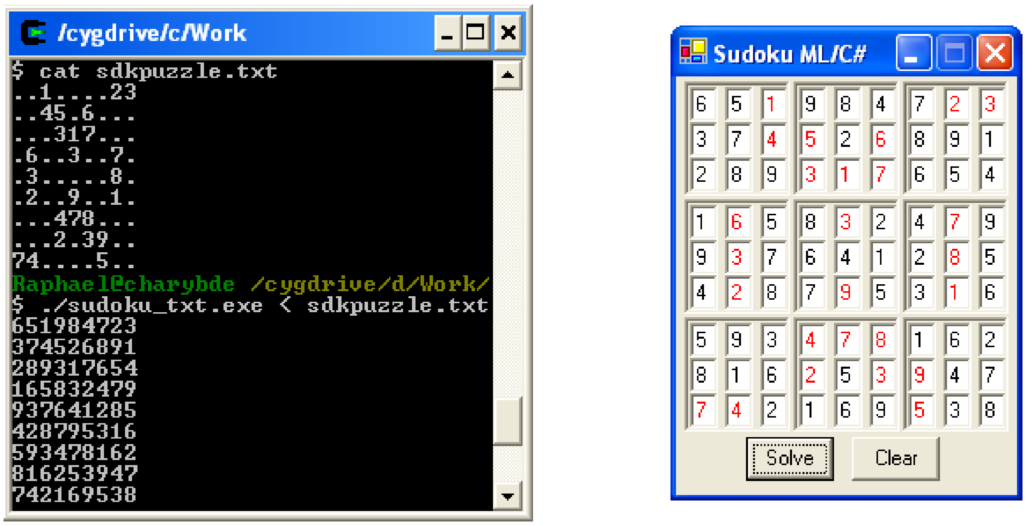


FIG. 5.1 – *Solveurs de Sudoku : grille dans un fichier texte à gauche, et saisie dans l'interface graphique à droite*

Nous munissons ce programme d'une interface graphique écrite en C# : les flux d'entrées-sorties sont redirigés en employant la technique décrite plus haut. Voici le code C# appelé lorsque l'utilisateur appuie sur le bouton `Solve` :

```
private void SolveHandler(object sender, EventArgs e) {
  //lecture de la grille d'entrée
  byte[] bytes = System.Text.ASCIIEncoding.Default.GetBytes(GetGrid());
  System.IO.Stream mlin = new MemoryStream(bytes);//surcharge entrée
  CamIL.IO.set_stdin(mlin);
  System.IO.Stream mlout = new MemoryStream(100); //surcharge sortie
  CamIL.IO.set_stdout(mlout);
  MLTop.startup();                                //appel du programme Caml
  mlout.Seek(0,SeekOrigin.Begin);
  StreamReader sr = new StreamReader(mlout);
  SetGrid(sr.ReadToEnd());                        //affichage du résultat
}
```

La classe `MemoryStream`, faisant partie de la BCL, hérite de la classe `Stream` : elle permet d'implanter un flot (en lecture ou en écriture) à partir d'une zone de mémoire. Tous les appels de la librairie standard de OCamIL manipulant l'entrée et la sortie standard seront dirigés vers une instance de cette classe. La zone de mémoire est construite à partir d'une chaîne de caractères (et vice-versa) qui représente un état de la grille de Sudoku. Les méthodes C# `GetGrid` et `SetGrid` font le lien entre l'affichage graphique et cette chaîne de caractères.

La classe `MLTop` est située dans la bibliothèque compilée à partir des sources du programme original et contient les méthodes de démarrage du code Caml (voir la section 3.2.2.2) : il est tout-à-fait possible d'appeler ces méthodes depuis le code C# de l'interface graphique pour exécuter le code Caml du solveur.

Détourner le système de fichiers. L'idée précédente se généralise aux descripteurs de fichiers. La bibliothèque d'exécution de OCamIL définit deux types délégués qui permettent précisément de surcharger les fonctions Caml `open_descriptor_in` et `open_descriptor_out` du module `Pervasives`. Il s'agit pour la lecture de :

```
BinaryReader CustomOpenFileDescriptorIn(string filename, int32 filemode,
                                        int32 fileaccess, int32 fileshare)
```

et pour l'écriture de :

```
BinaryWriter CustomOpenFileDescriptorOut(string filename, int32 filemode,
                                         int32 fileaccess, int32 fileshare)
```

Pour la redirection des flux la bibliothèque d'exécution de OCamIL expose les méthodes suivantes :

- `void set_custom_openfile_in(CustomOpenFileDescriptorIn ci)`
- `void set_custom_openfile_out(CustomOpenFileDescriptorOut co)`

On peut trouver des exemples de telles surcharges dans la suite de ce chapitre.

Embarquer le contrôle `Graphics`. La bibliothèque `Graphics` de Caml constitue une forme d'entrée-sortie. Elle figure dans la distribution OCamIL sous forme d'une couche au dessus des contrôles Windows (espace de noms `System.Windows.Forms`) de la BCL[1].

Toutes les opérations graphiques sont réalisées sur une instance de la classe `CamILGraphics` qui hérite de `System.Windows.Forms.Control` (servant de base aux widgets .NET).

Par défaut, cet objet est intancié et recouvre une fenêtre (un objet de type `System.Windows.Forms.Form`) au moment de l'appel à la fonction `open_graph` de la bibliothèque `Graphics`, de manière fidèle à l'implantation de référence.

Ici aussi nous avons prévu un moyen de surcharger ce comportement, en permettant à un code externe d'enregistrer auprès de la bibliothèque `Graphics` une instance de `CamILGraphics` de son choix : cette instance est créée par le code externe et donc embarquée au sein d'autres contrôles de manière totalement libre. Dans ce cas l'appel à la fonction `open_graph` n'induit pas l'instanciation d'une nouvelle fenêtre.

Un exemple complet : l'auto-assemblage. Nous avons choisi pour cet exemple un programme Caml écrit par Fabien Tarissan dans le cadre de sa thèse de bio-informatique au laboratoire PPS, et servant d'illustration à ses travaux sur l'auto-assemblage de graphes [29].

Le programme permet :

- D'éditer la description de graphe(s) cible(s) ainsi que les sous-graphes autorisés dans une construction. Les structures ainsi décrites sont sauvées sur le disque à un format propriétaire (la sérialisation directe des types Caml utilisés en interne).

1. Merci à Clément Capel pour sa contribution à l'implantation de la bibliothèque `Graphics`.

– De générer les règles d'interactions en utilisant l'algorithme décrit dans les travaux de Fabien Tarissan et de suivre sur un affichage graphique l'évolution d'un groupe d'agents dont le comportement est dicté par ces règles. Le programme Caml utilise la bibliothèque `Graphics`.

Le programme original est disponible à l'adresse suivante : `http://www.pps.jussieu.fr/~tarissan/self/implem_sag.tar.gz`. À côté des sources se trouvent des fichiers GIF animés montrant le déroulement du programme sur des exemples.

Plusieurs choix sont possibles pour munir le programme d'une interface graphique à base de menus et de boutons, basée sur les contrôles .NET de l'espace de noms `System.Windows.Forms`.

– Nous pouvons directement implanter cette interface en Caml, en exposant les éléments utiles de la bibliothèque `Forms` au moyen de classes Objective Caml en passant par notre outil OJacaré.NET : la logique de l'interface est alors codée en Caml.
– Nous pouvons coder l'interface en C# et nous reposer sur des voies de communication élémentaires entre les composants C# et Caml, utilisant les mécanismes de surcharge d'entrées-sorties exposés dans cette section.

Pour cela, nous avons adapté la boucle principale du programme pour qu'elle interprète des commandes reçues sur l'entrée standard. Ensuite, nous avons programmé une interface graphique en C# qui envoie ces commandes sur une surcharge de l'entrée standard lorsque les boutons ou menus sont actionnés. Il est possible de charger un fichier de description de graphe à partir du disque dur (auquel cas une fenêtre de sélection apparaît) ou parmi une liste de graphes prédéfinie. Dans les deux cas nous avons utilisé une surcharge des entrées de fichiers.

Enfin, l'interface graphique prévoit une zone d'affichage qui s'enregistre auprès de la bibliothèque graphique de OCamIL (voir la figure 5.2).

5.1.1.2 OJacaré.NET : exploitation des ressources .NET

La BCL fournit déjà un grand nombre de services qu'il est intéressant de pouvoir exploiter depuis des programmes Caml. Ces services sont interfaçables facilement au moyen de OJacaré.NET.

Un exemple : outils XML. Nous détaillons un exemple d'utilisation de certaines des classes fournies par l'espace de noms `System.Xml`. Pour cela nous revenons à l'application d'auto-assemblage de graphes présentée à la section 5.1.1.1.

Les fichiers décrivant les graphes d'assemblage étaient à un format propriétaire dans l'application d'origine. Nous voulons utiliser un format ouvert, ce qui permet de générer et éditer ces fichiers en dehors du programme Caml et nous choisissons pour cela un format `XML`.

Nous avons tout d'abord défini un schéma XSD [88] définissant la grammaire des fichiers de graphes (figure 5.3 : les feuilles de l'arbre sont des entiers ou des booléens).

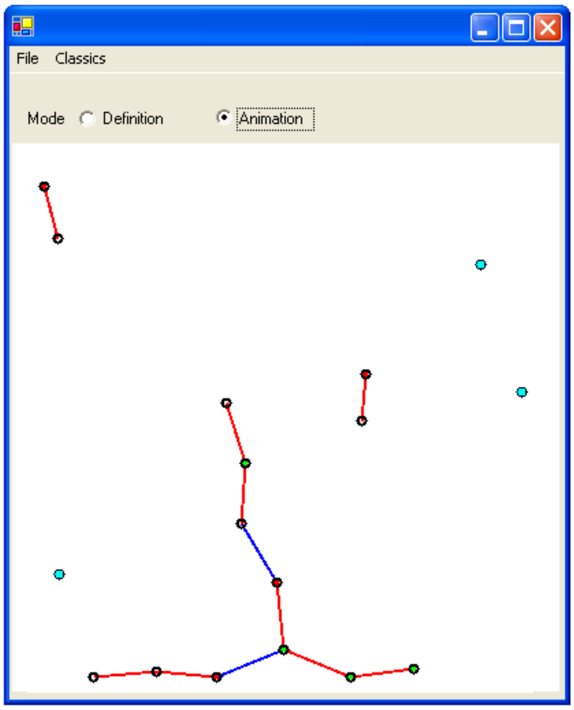


FIG. 5.2 – *Auto-assemblage avec interface graphique*

Bien que la bibliothèque standard de Caml ne propose pas de service de manipulation de fichiers `XML`, il est simple de combler ce vide en faisant notre choix entre les multiples classes consacrées à ce format au sein de la BCL et d'utiliser OJacaré.NET pour les utiliser dans le programme. Voici le fichier IDL utilisé :

```
package [assembly mscorlib] System.IO;

class TextWriter {}

class StringWriter extends TextWriter {
  [name string_writer] <init>();
  [name to_string] string ToString();
}

package [assembly System.Xml] System.Xml;

class XmlTextWriter {
  [name xml_text_writer] <init>(System.IO.TextWriter);
  [name close] void Close();
  [name flush] void Flush();
  [name write_start_document] void WriteStartDocument();
  [name write_end_document] void WriteEndDocument();
  [name write_start_element] void WriteStartElement(string);
  [name write_end_element] void WriteEndElement();
  [name write_element_string] void WriteElementString(string,string);
  [name write_attribute_string] void WriteAttributeString(string,string);
}

class XmlNode {
```

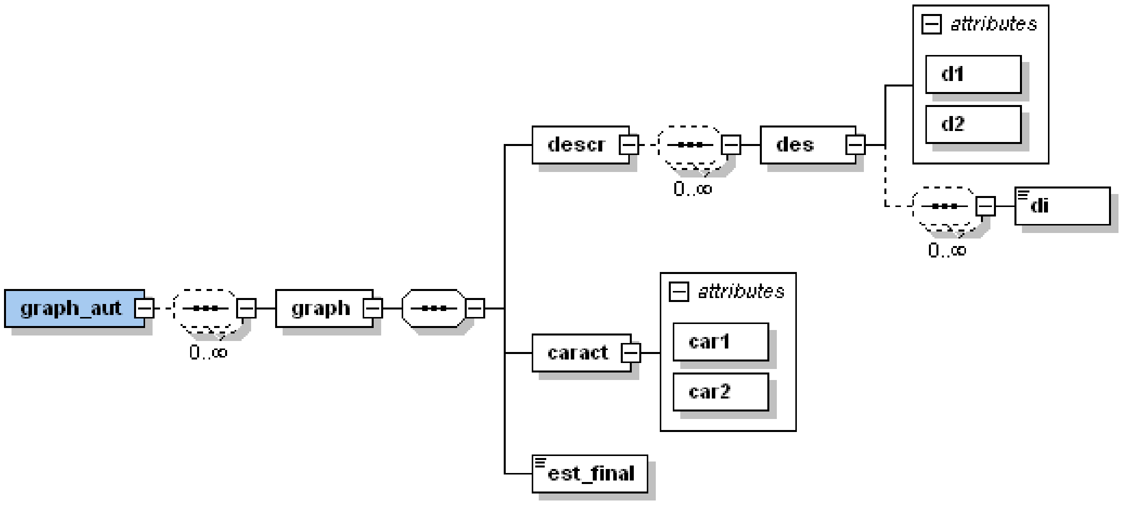


FIG. 5.3 – *Schéma XSD pour les descriptions de graphes*

```
  [name select_single_node] XmlNode SelectSingleNode(string);
  [name select_nodes] XmlNodeList SelectNodes(string);
  [name inner_text]string get_InnerText();
}

class XmlNodeList {
  [name count] int get_Count();
  [name item] XmlNode get_ItemOf(int);
}

class XmlDocument extends XmlNode {
  [name xml_document] <init>();
  [name load] void Load(string);
  [name load_xml] void LoadXml(string);
}
```

Nous omettons ici le code nécessaire à la validation des fichiers XML face à leur schéma. Les classes `StringWriter` et `XmlTextWriter` sont utilisées pour sauvegarder les structures externes sous forme XML, alors que les classes `XmlNode`, `XmlNodeList` et `XmlDocument` permettent de parcourir un document XML et de remplir les structures internes. Voici un extrait du programme OCamIL d'auto-assemblage contenant le code de lecture reposant sur les classes engendrées par OJacaré.NET.

```
let read_des node =
  let nodes = node#select_nodes "di" in
  let intl = ref [] in
    for i = 0 to (nodes#count()) -1 do
      intl := (int_of_string ((nodes#item i)#inner_text()))::!intl
    done;
    let d1 = int_of_string ((node#select_single_node "@d1")#inner_text()) in
    let d2 = int_of_string ((node#select_single_node "@d2")#inner_text()) in
      (List.rev !intl, (d1,d2));;
```

```
let read_descr node =
  let nodes = node#select_nodes "des" in
  let nb = nodes#count() in
  let tab = Array.create nb ([],(0,0)) in
    for i = 0 to nb - 1 do
       tab.(i) <- read_des (nodes#item i)
    done;
    tab;;

let read_caract node =
  let c1 = int_of_string ((node#select_single_node "@car1")#inner_text()) in
  let c2 = int_of_string ((node#select_single_node "@car2")#inner_text()) in
  (c1,c2)

let read_ef node =
  bool_of_string (node#inner_text ());;

let read_graph node =
  let d = read_descr (node#select_single_node "descr") in
  let c = read_caract (node#select_single_node "caract") in
  let ef = read_ef (node#select_single_node "est_final") in
  {descr = d; caract = c; est_final = ef};;

let read name =
  try
    let doc = new xml_document () in
      doc#load name;
      let graphs = doc#select_nodes "//graph" in
      let gal = ref [] in
        for i = 0 to (graphs#count()) -1 do
           gal := (read_graph (graphs#item i))::!gal
        done;
        List.rev !gal
  with _ -> []
```

Ce code fait une utilisation des requêtes XPath pour le moins élémentaire...

Remarque : En dehors de l'exemple présenté ci-dessus, l'espace de noms `System.Xml` permet encore beaucoup de choses : par exemple les documents XML peuvent être manipulés au moyen du modèle DOM de niveau 2 [87] et les transformations XSL [86] sont prises en charge.

Autres ressources de la BCL. Sans vouloir être exhaustif, voici quelques autres services dignes d'intérêt :
- Liaison aux bases de données. L'espace de noms `System.Data` permet de programmer des clients `SQL`, `ODBC`, `Oracle` et `OLE DB` et introduit des structures de données adaptées aux bases de données.
- Services réseaux. L'espace de noms `System.Net` permet de manipuler simplement la plupart des protocoles réseaux.
- Applications distribuées. L'espace de noms `System.Runtime.Remoting` fournit des services de sérialisation et d'invocation d'objets distants.

- Cryptographie. La plate-forme implante des standards tels que `CMS` (*Cryptographic Message Syntax*) et `PKCS #7` (*Public-Key Cryptography Standards #7*).
- Génération de code. Il est possible d'engendrer du code CIL directement ou bien de manipuler un objet `CodeDom` de plus haut niveau, pouvant être imprimé en différents langages (C#, Visual Basic.Net, J# etc...).

5.1.1.3 Liens avec l'univers natif (non géré)

Utilisation de composants COM. Une contrainte primordiale dans le cahier des charges de la plate-forme .NET était la compatibilité avec la technologie COM de Microsoft, fournisseur historique de composants logiciels sur la plate-forme Windows. Il est de fait possible de construire des composants COM à partir de composants .NET ainsi que d'utiliser des composants COM depuis .NET, les services d'interopérabilité se chargeant des opérations de sérialisation.

- Un composant COM peut être utilisé depuis un client .NET en passant par la génération automatique d'une enveloppe gérée se chargeant d'exposer les interfaces, types et événements et de sérialiser appels de méthodes et exceptions. Cette enveloppe est générée par l'outil `tlbimp.exe` (*Type Library Importer*, fourni avec le kit de développement .NET), qui inspecte la bibliothèque de types du composant COM et génère un assemblage .NET d'enveloppe.
- Il est également possible d'implanter des composants COM dans un langage de la plate-forme .NET. Quelques opérations permettent d'exposer des assemblages .NET au monde COM : l'outil `tlbexp.exe` (*Type Library Exporter*) génère une bibliothèque de types COM à partir des méta-données contenus dans l'assemblage .NET et l'outil `regasm.exe` (*Register Assembly*) réalise l'enregistrement du composant dans la base de registres. Tous deux sont fournis avec le kit de développement de .NET. La génération de la bibliothèque de types par `tlbexp.exe` peut-être dirigée par le code source, non pas au moyen d'un fichier IDL comme c'est traditionnellement le cas dans le monde COM, mais par des attributs disséminés dans le code source, ce qui permet de spécifier quelles classes/méthodes doivent être exportées, de spécifier explicitement les différents GUID, de donner des précisions sur la sérialisation, etc...

De nombreux détails sur l'interopération entre .NET et composants COM sont donnés dans l'article en ligne [12].

Certaines grosses applications Windows munies d'une API COM, telles que les applications de la suite Office, Visual Studio ou encore DirectX, sont fournies avec des PIA (*Primary Interop Assemblies*) : il s'agit d'assemblages enveloppant les composants COM qui sont spécialement optimisés, et plus efficaces que les enveloppes générées automatiquement par l'outil `tlbimp.exe`.

Voici par exemple l'appel de méthodes de DirectSound permettant la lecture d'un fichier *Wave* depuis un composant .NET. Afin de rendre accessible au code

Caml quelques-uns des éléments du PIA, nous utilisons OJacaré.NET avec le fichier `directsound.idl` suivant :

```
package [assembly System.Windows.Forms] System.Windows.Forms;

class Control {}

class Form extends Control {
  [name form]<init>();
}

class Application {
  [name application__run]static void Run(Form);
}

package [assembly Microsoft.DirectX.DirectSound] Microsoft.DirectX.DirectSound;

enum CooperativeLevel {
  [name CLNormal]Normal = 1
}

enum BufferPlayFlags {
  [name BPFDefault]Default = 0,
  [name BPFLooping]Looping = 1
}

class Device {
  [name device]<init>();
  [name set_cooperative_level]
      void SetCooperativeLevel(System.Windows.Forms.Control,CooperativeLevel);
}

class Buffer {
  [name buffer]<init>(string,Device);
  [name play]void Play(int,BufferPlayFlags);
}
```

Le code Caml minimaliste qui suit permet de jouer un son (notons que l'objet de base, un *device*, doit être attaché à une fenêtre Windows, d'où la nécessité d'ouvrir un formulaire).

```
open Directsound

let _ =
  let window = new form () in
  let th = Thread.create application__run window in
  let dev = new device () in
    dev#set_cooperative_level (window :> csControl) CLNormal;
    let buffer = new buffer "Audio.wav" dev in
      buffer#play 0 BPFDefault;;
```

Appel de code natif C. Le lien direct avec C que permettent les déclarations `external` de Caml sont perdues avec OCamIL puisque celles-ci permettent désormais l'appel de code CIL. Il est toutefois encore possible d'utiliser du code « non géré » depuis les programmes OCamIL en passant indirectement par des wrappers écrits dans un autre langage .NET, que ce soit un langage purement géré comme C# (auquel cas on utilise explicitement le mécanisme `P/Invoke`) ou bien `Visual C++` avec extensions gérées, qui permet le mélange de code géré et non géré dans un même fichier source.

5.1.2 Construction d'appliquettes

5.1.2.1 Principe

La plate-forme .NET propose l'équivalent des appliquettes Java : embarquer des composants .NET munis de fonctionnalités graphiques dans Internet Explorer, et s'exécutant du côté du client.

Composants `Winform` dans `Internet Explorer`. Toute classe héritant de la classe `Windows.Forms.Control` et définie dans un assemblage de bibliothèque peut potentiellement être embarquée dans `Internet Explorer`. L'assemblage, qui peut ne pas être signé, est situé sur le serveur. Il est téléchargé sur le poste client quand le navigateur affiche une page contenant les informations de contrôle de la forme suivante :

```
<html>
  <head>
    <link rel='Configuration'
          href='http://some.url.com/myAssembly.dll.config'>
    </link>
  </head>
  <body>
    ...
    <object id='myControl'
            classid='http:myAssembly.dll#MyNamespace.MyClass'
            height=200 width=200>
    </object>
    ...
  </body>
</html>
```

La balise `<object>` est déjà utilisée dans le monde Windows pour embarquer des composants ActiveX. Ici le GUID (*Globally Unique Identifier*) de l'attribut `classid` est remplacé par une chaîne de caractères qui indique à la fois la localisation de l'assemblage sur le serveur et le nom de la classe CTS du contrôle à instancier (celle-ci doit être publique et avoir un constructeur public).

La balise `<link>` est optionnelle. Elle permet de localiser un fichier de configuration utilisé par le chargeur de classes du CLR pour trouver les dépendances de l'assemblage contenant le contrôle. Ces dépendances peuvent tout-à-fait se trouver elles aussi sur le serveur.

En dehors de la page `http://msdn.microsoft.com/msdnmag/issues/02/06/Rich`, il existe peu de documentation pratique sur les appliquettes .NET, aussi il faut veiller à prendre garde aux points suivants :

- Dans le cas où les assemblages utilisés sont signés, ils doivent être compilés avec un attribut spécial : `AllowPartiallyTrustedCallersAttribute` pour pouvoir les exécuter depuis une zone de sécurité restreinte (ce que l'on doit toujours supposer dans le déploiement d'une appliquette, puisque la zone internet n'est pas considérée comme sûre dans les réglages par défaut de Windows).
- Les différents fichiers mis en jeu doivent avoir les droits de lecture et d'exécution suffisants, et les fichiers d'assemblages doivent être servis avec un type MIME reconnu, tel que `x-msdownload`. Si ces différents pré-requis ne sont pas respectés, le composant ne fonctionnera pas sans pour autant qu'un quelconque avertissement donne une indication sur le problème.
- Les assemblages téléchargés sont stockés localement dans un cache spécial appelé « Assembly Download Cache ». En phase de développement il peut être nécessaire de vider ce tampon (commande `gacutil /cdl`).

Pour l'instant, seul Internet Explorer est capable d'embarquer des composants .NET, sous Windows. Le projet Mono travaille d'arrache-pied à l'adaptation des classes de l'espace de noms `Windows.Forms` : lorsque celle-ci sera complète, on pourra espérer exécuter des contrôles .NET dans d'autres navigateurs et sous d'autres systèmes d'exploitation.

Questions de sécurité. Un composant OCamIL distribué sous forme d'appliquette s'exécute dans un environnement aux droits limités. En particulier les accès disques traditionnels ne sont pas permis sans modifier les paramètres de sécurité de Windows.
Plusieurs possibilités existent selon la nature des accès disques souhaités ; elles se réalisent toutes en court-circuitant le système de fichiers usuel au moyen des fonctions `set_custom_openfile_in` et `set_custom_openfile_out` :

- L'accès en lecture à des fichiers situés sur le serveur est toujours possible : on utilise pour cela les classes `HttpWebRequest` et `HttpWebResponse` de l'espace de noms `System.Net`. En cas de succès de la requête `HTTP`, cette dernière donne accès à un objet `Stream`. La solution équivalente en cas d'accès en écriture passe par l'envoi de fichiers au serveur par `HTTP` (ce qui suppose la programmation d'un service de fichiers du côté du serveur, utilisant une base de données par exemple).
- La classe `OpenFileDialog` de l'espace de noms `System.Windows.Forms` peut être utilisée dans un environnement de sécurité contraignant. Celle-ci permet d'afficher une boîte de dialogue demandant à l'utilisateur de sélectionner un fichier sur le disque local. La classe donne ensuite accès à un `Stream` permettant lecture et/ou écriture, sans donner aucune autre information sur le fichier lui-même. C'est parce que cette façon de procéder met en jeu l'utilisateur et fait en sorte que la localisation du fichier reste opaque pour le programme qu'elle est autorisée pour les appliquettes. Elle est raisonnable dans le cas où l'appliquette a le rôle d'un service éditant des fichiers de l'utilisateur.
- Lorsque l'appliquette doit sauvegarder des informations d'une ouverture sur

l'autre, les solutions précédentes peuvent s'avérer trop lourdes. La BCL développe une notion de stockages isolés (*Isolated Storage*), qui permet à des applications d'avoir un accès complet à des données sur le poste client d'une façon qui abstrait la représentation et la localisation réelle de ces données (manipulation de *Stores* au lieu de fichiers) et isole ces données entre applications distinctes. Les implantations des stockages isolés prennent la forme de classes dérivant de `System.IO.IsolatedStorage`. Tout en s'appuyant sur le système de fichiers local, via des répertoires temporaires administrés par la plate-forme .NET, les stockages isolés définissent un cadre opaque et sûr pour la manipulation des fichiers. Les données peuvent être cloisonnées entre utilisateurs et assemblages différents, mais aussi par domaines d'applications. Un administrateur d'un poste peut limiter la taille des stockages isolés, les vider etc...

- Une manière commode de charger des données accessibles en lecture seule est de les inclure directement sous formes de ressources dans l'assemblage (celui-là même qui définit le contrôle graphique ou bien l'une de ses dépendances). Cette inclusion est réalisée à l'édition de liens des composants : chaque fichier est alors identifié au moyen d'une clef, et rendu accessible au composant .NET au moyen d'un `Stream`. Cette solution facilite grandement le déploiement puisque les chargements des assemblages et des ressources sont alors indissociables.

5.1.2.2 Exemples

Voici des appliquettes construites à partir de certains des exemples précédents.

L'auto-assemblage. L'application d'auto-assemblage de graphes présentée à la section 5.1.1.1 existe également sous forme d'appliquette. Pour cela nous utilisons une version légèrement différente de l'interface graphique en C# : la sélection de fichiers sur le disque a disparu et la sélection des graphes standard ne se fait plus par un menu mais par un composant liste (voir la figure 5.4).

La redirection des entrées de fichiers est différentes puisqu'elle utilise des requêtes HTTP pour aller chercher les fichiers XML de description de graphes directement sur le serveur Web.

Le traceur de rayons. Suivant les mêmes principes, on peut adapter le traceur de rayons de la section 4.2.3.2 (figure 5.5).

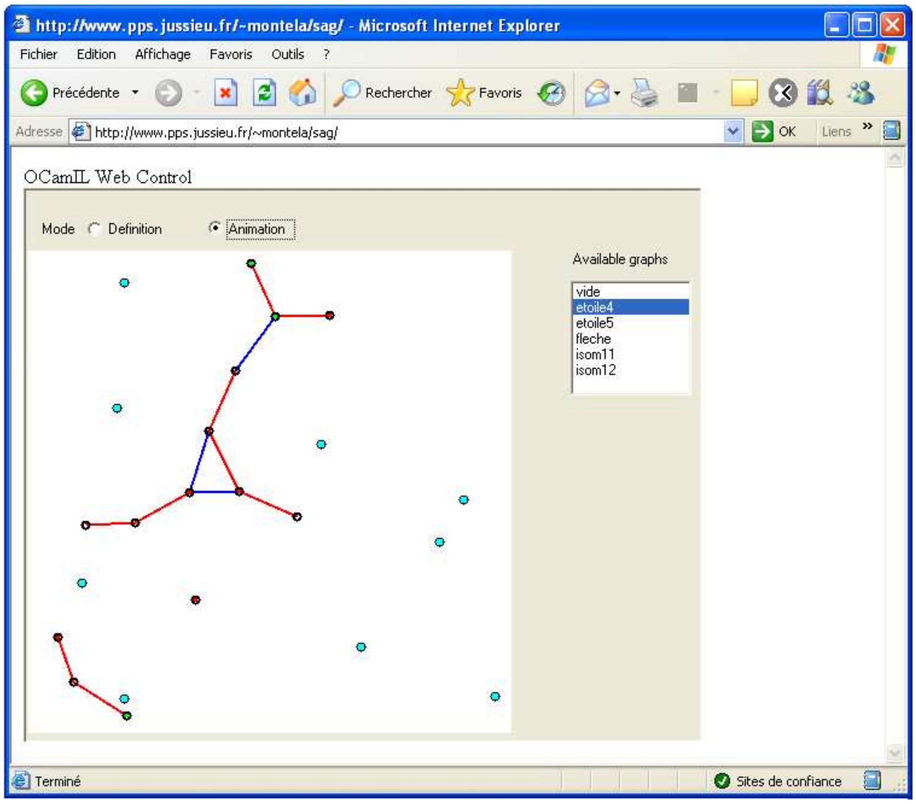


FIG. 5.4 – *Auto-assemblage : appliquette*

5.2 Applications liées au compilateur OCamIL

Nous présentons dans cette deuxième section les différentes réalisations faisant partie de la distribution OCamIL.

5.2.1 Bootstrap et toplevel

5.2.1.1 Bootstrap

Nous décrivons ici les différentes étapes qui mènent des sources de OCamIL jusqu'à un compilateur bootstrappé qui tourne dans l'environnement .NET ; celles-ci sont décrites sur la figure 5.6.

La compilation de OCamIL nécessite le compilateur de bytecode Objective Caml d'origine (`ocamlc`) ainsi que la machine virtuelle associée (notée μ). Sur la figure, `mlB` dénote le bytecode Objective Caml d'origine.

Étapes de construction (en suivant la figure 5.6):

1. Nous commençons par compiler un compilateur hybride `pre-ocamil` capable de produire des exécutables et des bibliothèques CIL à partir de fichiers sources Objective Caml. Remarquons cependant que ce compilateur s'exécute dans l'environnement Objective Caml standard.

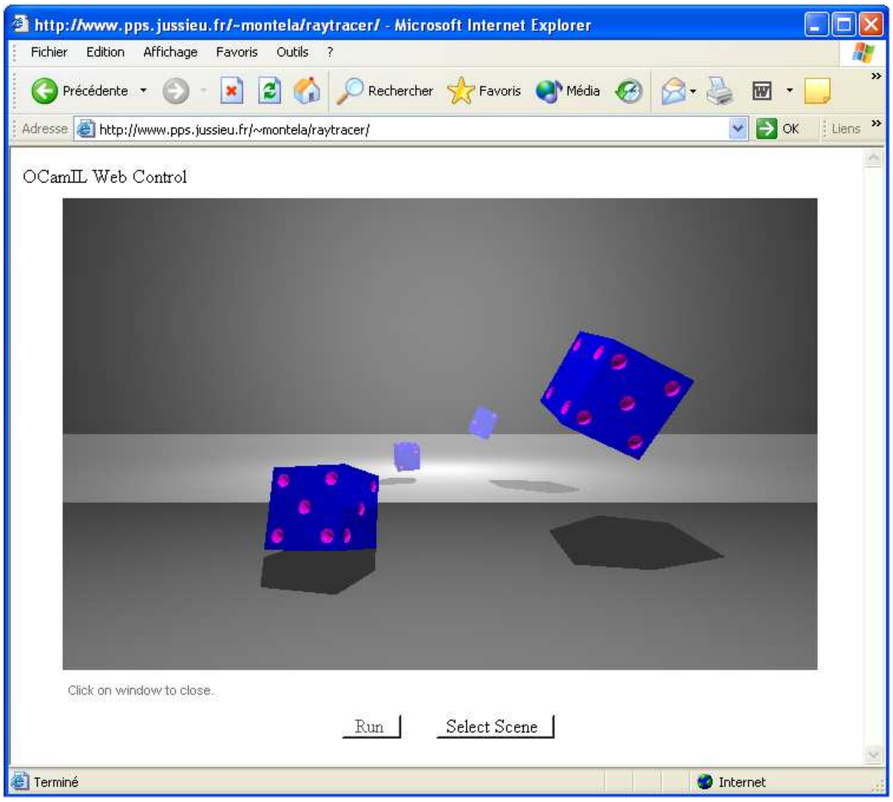


FIG. 5.5 – *Appliquette de lancer de rayons Objective Caml*

2. Nous recompilons ensuite les sources de OCamIL au moyen de ce compilateur fraîchement obtenu. Cela produit `ocamil`, lui-même un exécutable en bytecode .NET.

À ce moment-là nous n'avons plus besoin du système Objective Caml ni du compilateur `pre-ocamil`.

Bootstrap (en suivant la figure la figure 5.6): nous nous servons du dernier compilateur obtenu pour se compiler lui-même. Il nous faut deux cycles afin d'atteindre un point fixe (`ocamil-2` est identique à `ocamil-3`) à cause des différences minimes de sémantique opérationnelle exposées dans la section 3.1.3.1 : elles ont pour conséquence de modifier l'ordre de génération de code lors de la compilation de OCamIL. Ici cela n'affecte pas la sémantique du programme résultant mais seulement la disposition physique de son code. Le cycle supplémentaire élimine cette différence.

Remarque : même si le compilateur `pre-ocamil` est substituable au compilateur `camil` dans la grande majorité des cas, le fait qu'il tourne dans un environnement différent des programmes qu'il construit peut engendrer des problèmes, le plus significatif étant que les formats de sérialisation de données ne sont pas compatibles entre Objective Caml et OCamIL (les fonctions de sérialisation de ce dernier étant directement implantées au dessus de l'API de sérialisation de la BCL). Cela signifie que `pre-ocamil` n'est pas capable de lire des données sérialisées par les programmes

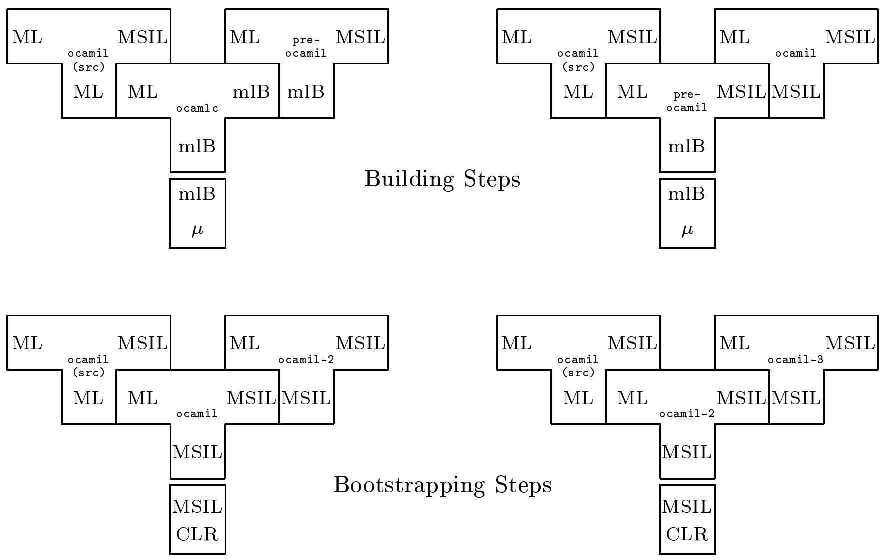


FIG. 5.6 – *Étapes de construction et de bootstrap*

qu'il a compilés, situation qui peut se produire quand on compile directement depuis un arbre de syntaxe abstrait sérialisé à la place d'un fichier source (comme engendrés par les pré-processeurs) ou dans le cas d'édition de liens dynamique : dans ces cas on ne pourra pas utiliser `pre-ocamil`.

De plus comme les fichiers `cmi`/`cmx` contiennent des données sérialisées, la différence de formats implique que les fichiers de bibliothèques compilés avec `pre-ocamil` ne peuvent être utilisés avec `ocamil`. Ils doivent être recompilés par `ocamil` lui-même.

5.2.1.2 Application : le toplevel

Après bootstrap le compilateur OCamIL tourne dans le même environnement que les exécutables qu'il produit. En utilisant les possibilités offertes par la plate-forme .NET en matière de génération et d'exécution dynamique de code, il est possible de construire un toplevel `ocamiltop`. Un toplevel compile des déclarations Objective Caml et les exécute à la volée, tout en gérant une table des symboles.

La figure 5.7 montre les composants du toplevel et la manière dont ils opèrent pour compiler une expression Objective Caml.

Organisation du toplevel. Le toplevel maintient une table d'association entre les variables liées à toplevel et leurs valeurs. Deux fonctions se chargent respectivement des accès en lecture (`Toploop.getvalue`) et en l'écriture (`Toploop.setvalue`) dans cette table. La boucle d'interaction répète la séquence suivante :

1. Le moteur du toplevel consomme une expression Objective Caml $phrase_n$.
2. *Compilation vers CIL.* Le toplevel utilise le moteur du compilateur `ocamil` pour compiler l'expression vers du code CIL ; l'expression passe par les phases habituelles d'analyses lexicale et syntaxique et d'inférence de types : grâce à la réutilisation du code du compilateur Caml ces phases sont rigoureusement

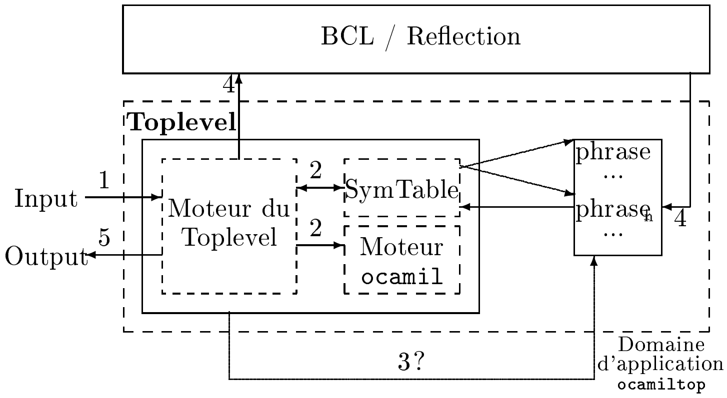


FIG. 5.7 – *Le moteur du toplevel*

identiques à celles du toplevel original. L'expression Caml ne peut être compilée telle quelle pour interagir avec le code du toplevel, elle est au préalable décorée par des termes Caml auxiliaires :

- on rajoute au code engendré deux fonctions permettant de raccrocher le code de la phrase proprement dite à la table d'association du toplevel ; pour cela une fonction `$define_getvalue` enregistre la fonction du toplevel `Toploop.getvalue` et une fonction `$define_setvalue` fait de même pour la fonction `Toploop.setvalue` : ces deux fonctions attendent une valeur de type `CamIL.Closure` qu'elles stockent dans des variables accessibles au code de la phrase,
- les variables libres sont remplacées le cas échéant par des appels à la fonction enregistrée par `$define_getvalue`,
- si la phrase lie une ou plusieurs variables à des valeurs, les appels adéquats à la fonction enregistrée par `$define_setvalue` sont réalisés : si une expression est évaluée sans lier de variable, son résultat est tout de même enregistré dans la table d'association via un symbole spécial `$prev` (*valeur précédente*),
- l'expression de la phrase à évaluer est encapsulée dans une fonction `$runme`.

3. *Génération physique du code.* Le code CIL est transformé vers une forme exécutable. Nous verrons que le détail de cette phase peut varier.
4. *Exécution du code dynamique.* Cette phase se déroule en trois étapes, qui font intervenir des services de l'API `Reflection` :
   - le code sous sa forme exécutable est initialisé,
   - les deux fonctions d'enregistrement sont appelées en leur passant en argument des références sur `Toploop.getvalue` et `Toploop.setvalue` : on établit ainsi le pont entre le code du toplevel et le code fraîchement engendré,
   - la fonction `$runme` est exécutée.
5. *Traitement du résultat.* Une fois le flot d'exécution revenu à la boucle du toplevel, celle-ci se charge de la sortie (typiquement en affichant les valeurs calculées).

Implantation élémentaire. Il a été rapide d'écrire un prototype de toplevel utilisant le disque local comme intermédiaire (cette version est décrite dans l'article [22]). Ce prototype fonctionne de la manière suivante (suivant la figure 5.8) :

- La génération physique du code consiste à générer un assemblage temporaire sur le disque local pour chaque expression saisie dans le toplevel. Outre la classe `MLTop` contenant les méthodes d'initialisation de l'assemblage, celui-ci ne contient qu'un module (un espace de noms pour OCamIL comportant les classes engendrées par la compilation de l'expression du toplevel, y compris celles rajoutées par la décoration (`$define_getvalue`, `$define_setvalue` et `$runme`). Comme cet espace de noms peut aussi contenir une classe utilisateur définissant un type algébrique Caml, l'éditeur de liens doit prendre garde à résoudre correctement vers cet assemblage temporaire toutes les références ultérieur au type défini.
- Lors de la phase d'exécution, le moteur du toplevel commence par appeler `System.Reflection.Assembly::LoadFrom` pour charger l'assemblage fraîchement engendré au sein de son domaine d'application (en mémoire). Après cela, les appels successifs dans l'assemblage se font au moyen des fonctions de l'API `Reflection`, `Assembly::GetType` et `Type::InvokeMember`.

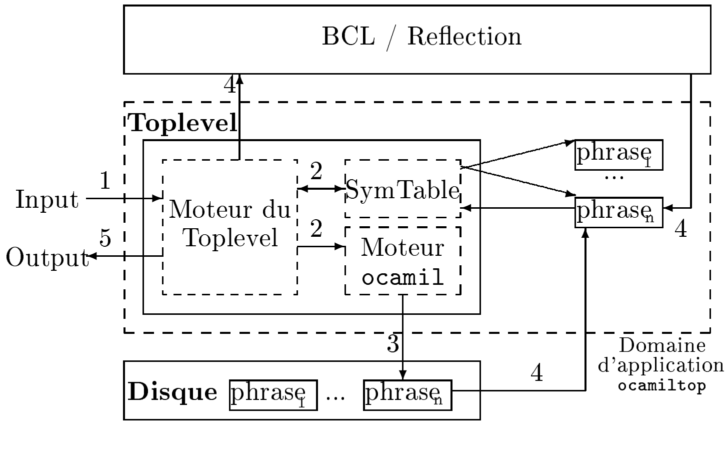


FIG. 5.8 – *Prototype du toplevel*

La multiplication des assemblages générés ne complique pas l'édition de liens (à part pour les types algébriques utilisateur) car les références ultérieures sont résolues via la table d'association du toplevel.

Il est vrai que passer systématiquement par le disque pour l'évaluation de chaque phrase est loin d'être optimal. Nous décrivons à la section 5.2.2.2 une deuxième implantation plus efficace et mieux intégrée.

Toplevel graphique. Il est possible de munir le toplevel d'une interface graphique en utilisant les techniques décrites à la section 5.1.1.1 : les entrées et sorties standard sont interfacées avec des flux personnalisés, reliés à des composants graphiques de saisie et d'affichage écrits en quelques lignes de C# (voir les figures 5.9 et 5.10).

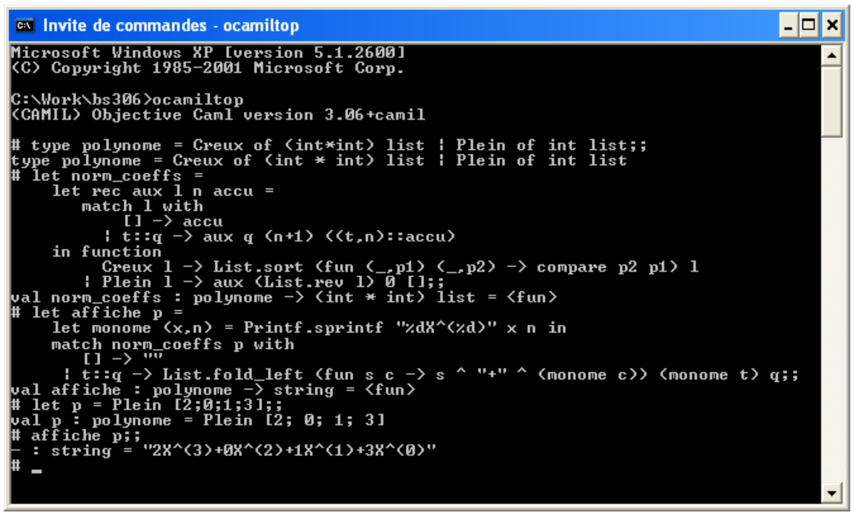


FIG. 5.9 – *Toplevel dans une fenêtre DOS*

Chargement dynamique de bibliothèques. Les bibliothèques de code Caml compilées par OCamIL peuvent être chargées dynamiquement dans deux cas de figure : le chargement dans une session du toplevel (directive `#load`) ou via le module `Dynlink`.

La figure 5.11 montre l'utilisation de la bibliothèque `Graphics` dans le toplevel : la bibliothèque est chargée dynamiquement.

OJacaré.NET dans le toplevel. Il est tout-à-fait possible d'utiliser des classes interfacées avec OJacaré.NET dans une session de toplevel, soit par l'évaluation directe du code produit par OJacaré.NET, soit par le chargement dynamique d'une bibliothèque pré-compilée à partir du code produit.

Nous illustrons ce mécanisme avec un exemple d'appel d'un service Web (protocole REST : XML via HTTP) depuis le toplevel. L'API de Yahoo Maps permet de récupérer des informations géographiques à un format XML sur une ville passée en argument.

Nous déclarons dans un même fichier `xmlhttp.idl` les services Web et XML de la BCL dont nous avons besoin (ici le format de Yahoo fait usage d'un espace de noms XML, si bien que nous utilisons des fonctions de lecture différentes de celles de la section 5.1.1.2).

```
package [assembly mscorlib] System.IO;

class Stream {}

class StreamReader {
  [name stream_reader]<init>(Stream);
  [name read_to_end]string ReadToEnd();
  [name close]void Close();
}
```

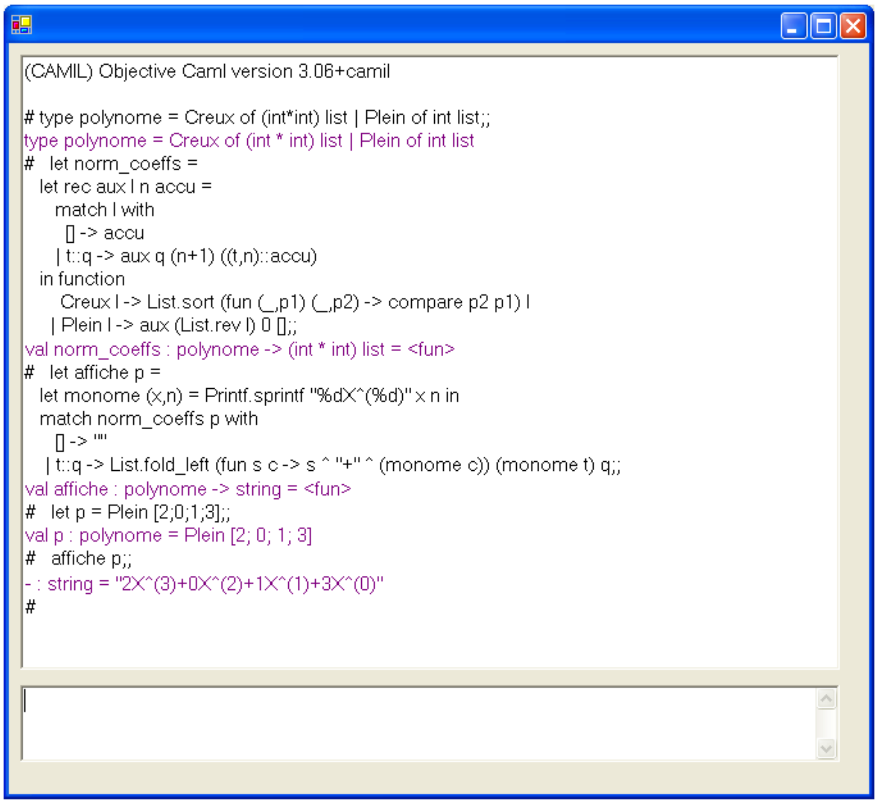


FIG. 5.10 – *Toplevel avec interface graphique*

```
package [assembly System] System.Net;

class WebRequest {
  [name webrequest__create]static WebRequest Create(string);
}

class WebResponse {}

class HttpWebRequest extends WebRequest {
  [name get_response]WebResponse GetResponse();
}

class HttpWebResponse extends WebResponse {
  [name get_response_stream]System.IO.Stream GetResponseStream();
}

package [assembly System.Xml] System.Xml;

class XmlDocument {
  [name xml_document] <init>();
  [name load_xml] void LoadXml(string);
  [name select_single_node_ns]
        XmlNode SelectSingleNode(string,XmlNamespaceManager);
  [name name_table] XmlNameTable get_NameTable();
}
```

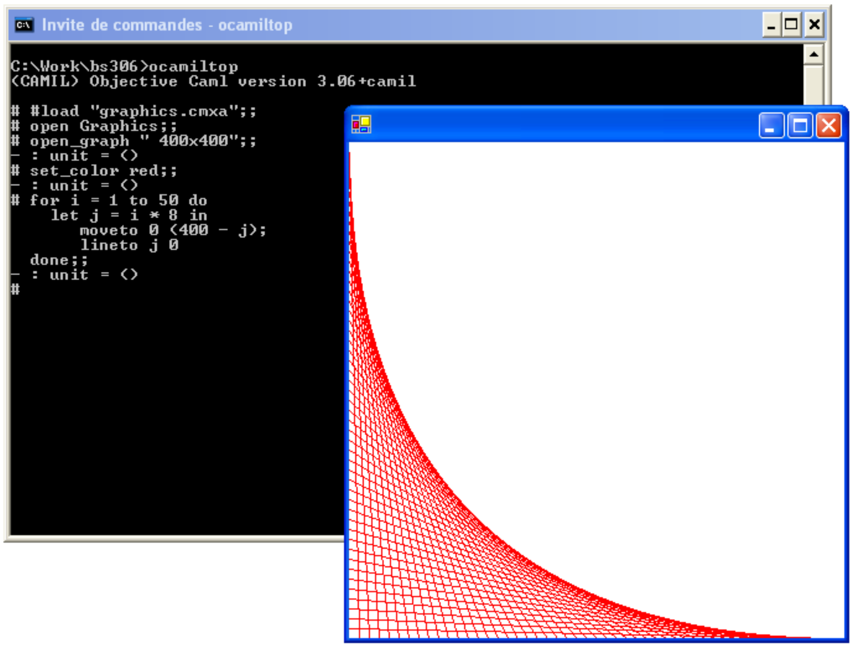


FIG. 5.11 – *Chargement d'assemblage dans le toplevel : exemple de* `graphics`

```
class XmlNamespaceManager {
  [name xml_namespace_manager]<init>(XmlNameTable);
  [name add_namespace]void AddNamespace(string,string);
}

class XmlNode {
  [name inner_text]string get_InnerText();
}

class XmlNameTable {}
```

Le fichier IDL est compilé par OJacaré.NET, et le résultat est compilé par OCamIL sous forme d'une bibliothèque. Cette bibliothèque est chargée et utilisée directement dans une session du toplevel, comme illustré sur la figure 5.12.

La directive `#load "xmlhttp.cmxa"` charge la bibliothèque compilée à partir du code engendré par OJacaré.NET sur le fichier `xmlhttp.idl`, ce qui autorise l'utilisation des classes correspondantes dans la session toplevel. On définit la fonction `get_yahoo_geoxml` qui effectue une requête au service Web de géolocalisation de Yahoo (on fabrique l'URL de la requête contenant un nom de ville) et retourne un XML de résultat. La fonction `get_lat_lon` extrait du fichier XML les coordonnées de la ville (latitude et longitude). Les requêtes se font ensuite simplement directement dans la session du toplevel.

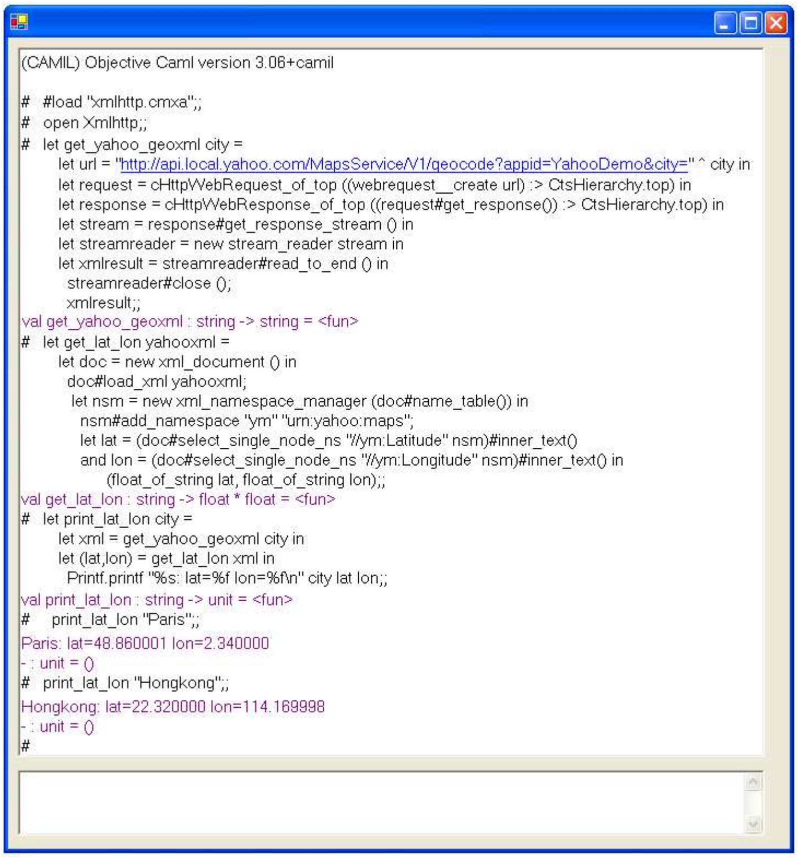


FIG. 5.12 – *Chargement dans le toplevel d'un assemblage généré par OJacaré*

5.2.2 Utilisation de OJacaré.NET au sein du compilateur

Cette section illustre l'application de OJacaré.NET à un problème concret et de grande échelle : interfacer l'API de `Reflection` avec le compilateur OCamIL et ainsi pouvoir bénéficier d'un nouveau générateur de code CIL, qui ne se base plus sur une commande externe `ilasm.exe`.

5.2.2.1 Back-end construit sur `Reflection.Emit`

Chaque assemblage .NET est formé des subdivisions suivantes :

- Un assemblage est constitué de modules,
- Un module définit des types (valeurs ou références),
- Chaque type définit des champs, des méthodes et des constructeurs,
- Le corps des méthodes et des constructeurs contient une suite d'instructions CIL.

L'API `Reflection.Emit` introduit des objets construisant les éléments de chacun de ces niveaux : les classes `AssemblyBuilder`, `ModuleBuilder`, `TypeBuilder`, `FieldBuilder`, `MethodBuilder`, `ConstructorBuilder` et `ILGenerator`.

La construction d'un assemblage complet procède par couches successives, tout objet d'un niveau pouvant référencer un objet d'un niveau supérieur (par exemple les champs peuvent recevoir un type en construction, le code des méthodes peut référencer des types ou des méthodes en cours de fabrication, etc)...

Dans la suite nous donnons des extraits du fichier IDL écrit pour OJacaré.NET et du code Caml ainsi interfacé avec les classes de `Reflection.Emit`, pour différentes étapes de la construction de l'assemblage.

Construction de l'assemblage et de son module. OCamIL se place dans un cadre où les assemblages produits ne sont formés que d'un seul module, ceci afin de n'avoir qu'un seul fichier à la sortie de l'édition de liens, comme c'est le cas pour le compilateur Caml standard.

```
// fichier reflection.idl (section : gestion des assemblages)
package [assembly mscorlib]System;

class AppDomain {
  [name define_dynamic_assembly] Reflection.Emit.AssemblyBuilder
      DefineDynamicAssembly(Reflection.AssemblyName,
                            Reflection.Emit.AssemblyBuilderAccess);
}

package [assembly mscorlib]System.Threading;

class Thread {
  [name thread_get_domain]static System.AppDomain GetDomain();
}

package [assembly mscorlib]System.Reflection.Emit;

enum AssemblyBuilderAccess {
  [name ABARun]Run = 1,
  [name ABASave]Save = 2,
  [name ABARunAndSave]RunAndSave = 3
}

class AssemblyBuilder extends System.Reflection.Assembly {
  [name define_dynamic_module] ModuleBuilder DefineDynamicModule(string, string);
  [name define_transient_dynamic_module]
     ModuleBuilder DefineDynamicModule(string, boolean);
  [name save] void Save(string);
  [name set_entrypoint] void SetEntryPoint(System.Reflection.MethodInfo);
  [name set_custom_attribute] void SetCustomAttribute(CustomAttributeBuilder);
}

class ModuleBuilder {
  [name define_type]
     TypeBuilder DefineType(string, System.Reflection.TypeAttributes);
  [name define_type_extends]
     TypeBuilder DefineType(string, System.Reflection.TypeAttributes,
                            System.Type);
}
```

Remarques :

- Il faut créer au départ un objet `AssemblyBuilder` à partir d'une instance d'un objet `AppDomain` représentant le domaine d'application courant, lui-même obtenu à partir d'une méthode statique de la classe `Thread`.
- Lors de son instanciation, il est décidé que l'objet `AssemblyBuilder` construira un assemblage voué à être exécuté, sauvé ou bien les deux (voir l'énumération `AssemblyBuilderAccess`).
- La classe `AssemblyName`, non détaillée ici, encapsule à la fois le nom de l'assemblage, une version et de manière optionnelle un nom fort (ces deux dernières informations sont construites à partir de classes définies dans l'espace de noms `Reflection`).
- Les modules construits peuvent être persistables ou temporaires : dans le premier cas il faut donner un nom logique et un nom de fichier alors que dans le deuxième cas un nom logique suffit (la méthode `DefineDynamicModule` attend alors un booléen qui indique s'il est nécessaire d'émettre les symboles avec le code).

Le code Objective Caml utilisant ces constructeurs est le suivant (le paramètre `icu` contient la description de l'unité de compilation dans la représentation intermédiaire IL de OCamIL) :

```
(* icu =  unité de compilation IL,
   fname = nom de fichier de l'assemblage émis ("" si pas de sauvegarde),
   keypairfile = clé pour signer l'assemblage ("" si pas de clé) *)

let ilcompunit icu fname keypairfile =
  let icuass = icu.icuass in (* structure décrivant l'assemblage)
  let assembly_access = if fname = "" then ABARun else ABASave in
  let domain = thread_get_domain () in
  let assbyname = new assembly_name () in
  assbyname#set_name icuass.anme;
  let version = new assembly_version (convert_version_format icuass.aver) in
  assbyname#set_version version;
    if keypairfile <> "" then begin    (* définir le nom fort si nécessaire *)
        let fs = file__open_read keypairfile in
        let sn = new strongname_keypair fs in
          fs#close();
          assbyname#set_keypair sn
    end;
  (* définition des constructeurs d'assemblage et de module *)
  let abuilder = domain#define_dynamic_assembly assbyname assembly_access in
  let mbuilder =
     if fname = "" then abuilder#define_transient_dynamic_module icuass.amod false
     else abuilder#define_dynamic_module icuass.amod fname
  in
  let entrypoint = ref None in
  (* géneration paresseuse des classes *)
  let classes_to_bake = List.map (ilclass entrypoint mbuilder) icu.icutypes in
  let type_builders, methods_to_bake =
    List.split (List.map (fun f -> f()) classes_to_bake)
  in
    (* génération des méthodes une fois toutes les classes définies *)
    List.iter (List.iter (fun m -> m())) methods_to_bake;
    (* finalisation des classes une fois leur contenu complet *)
    List.iter (fun tp -> let _ = tp#create_type() in ()) type_builders;
    (* définition du point d'entrée, récupéré par effet de bord en générant les méthodes *)
    match !entrypoint with
        Some minfo -> abuilder#set_entrypoint minfo
      | None -> ());
    if fname <> "" then abuilder#save fname     (* sauvegarde si nécessaire *)
```

On voit dans ce code la création des constructeurs d'assemblage et de module `abuilder` et `mbuilder`. L'étape suivante est la création des types (appels à la fonction `ilclass` détaillée dans la suite). Chaque appel à `ilclass` induit la construction d'un type CTS (mais sans encore construire les objets de l'étage inférieur) et retourne une fermeture permettant de différer les opérations de finalisations des objets construits :

- Une première passe permet la construction des champs et des méthodes, au moment où toutes les classes engendrées sont connues. On récupère alors une liste de types construits `type_builders` et une liste de méthodes à finaliser `methods_to_bake`.
- Lors d'une deuxième passe on applique les fermetures contenues dans la variable `methods_to_bake`, ce qui permet d'engendrer le code de chaque méthode au moment où toutes les classes, champs et déclarations de méthodes sont accessibles.
- On appelle la méthode `create_type` sur chaque type engendré, une étape réclamée par l'API `Reflection.Emit`.

Il ne reste plus alors qu'à définir le point d'entrée s'il existe (la valeur référencée par `!entrypoint` étant renseignée par effet de bord lors des opérations précédentes) et éventuellement à sauvegarder l'assemblage.

Création des types, champs et méthodes. Comme évoqué dans le paragraphe précédent, chaque création de type, champ ou méthode, est « paresseuse », au sens où l'objet est défini, mais pas complété tant que tous les autres objets qu'il peut référencer n'ont pas été eux-mêmes définis. Nous ajoutons les définitions suivantes au fichier IDL :

```
// fichier reflection.idl (section : construction des classes)
class TypeBuilder extends System.Type {
  [name create_type] System.Type CreateType();
  [name define_method]
     MethodBuilder DefineMethod(string, System.Reflection.MethodAttributes,
                                System.Reflection.CallingConventions,
                                System.Type, System.Type[]);
  [name define_constructor]
     ConstructorBuilder DefineConstructor(System.Reflection.MethodAttributes,
                                          System.Reflection.CallingConventions,
                                          System.Type[]);
  [name define_field] FieldBuilder DefineField(string, System.Type,
                                               System.Reflection.FieldAttributes);
}

class MethodBuilder extends System.Reflection.MethodInfo {
  [name get_il_generator] ILGenerator GetILGenerator();
}

class ConstructorBuilder extends System.Reflection.ConstructorInfo {
  [name get_il_generator] ILGenerator GetILGenerator();
}
```

```
class FieldBuilder extends System.Reflection.FieldInfo {}
```

Le constructeur de chaque catégorie d'objet attend un nom, des attributs spécifiques au type créé, et quand cela est nécessaire des références à d'autres objets créé, comme par exemple un type pour la construction d'un champ (mais à chaque fois l'objet correspondant est déjà connu car il fait partie d'un étage supérieur).

Les attributs spécifiques (tels que `MethodAttributes`, `CallingConventions`, etc...) sont définis par des énumérations combinables par un « ou » binaire. Voici un extrait de l'IDL les concernant :

```
// fichier reflection.idl (section : définition d'attributs)

flags MethodAttributes {
  [name RMAPrivate]Private = 1,
  [name RMAAssembly]Assembly = 3,
  [name RMAPublic]Public = 6,
  [name RMAStatic]Static = 0x10,
  [name RMAFinal]Final = 0x20,
  [name RMAVirtual]Virtual = 0x40,
  [name RMAAbstract]Abstract = 0x400,
  [name RMASpecialname]SpecialName = 0x800,
  [name RMARTSpecialname]RTSpecialName = 0x1000
}

flags FieldAttributes {
  [name RFAPrivate]Private = 1,
  [name RFAAssembly]Assembly = 3,
  [name RFAPublic]Public = 6,
  [name RFAStatic]Static = 0x10,
  [name RFAInitonly]InitOnly = 0x20
}

flags BindingFlags {
  [name BFInvokeMethod]InvokeMethod = 0x100
}

enum CallingConventions {
  [name RCCStandard]Standard = 1,
  [name RCCHasThis]HasThis = 0x20
}
...
```

Nous pouvons voir l'ensemble des classes ci-dessus interfacées dans le code Objective Caml qui suit :

```
(* mb = module builder, cl = définition de classes dans le langage IL,
   entrypoint permet de retourner le point d'entrée par effet de bord *)

let rec ilclass entrypoint mb cl =
  let qname = qualified_name cl.tdnsp cl.tdnme in
    (* définition du constructeur de type *)
    let typebuilder = match cl.tdext with
        None -> mb#define_type qname [RTAClass;RTAPublic]
      | Some tre ->  mb#define_type_extends qname [RTAClass;RTAPublic] (resolve_classtype tre)
    in
      (* ajout à la table des types définis *)
      Hashtbl.add transient_types qname (typebuilder :> cType);
      (* fonction effectuant la suite des opérations (sera appelée plus tard) *)
```

```
   (fun () ->
        List.iter (ilfield typebuilder qname) cl.tdfld;
        let methods_to_bake = List.map (ilmethod entrypoint typebuilder qname) cl.tdmet in
        (typebuilder, methods_to_bake))

(* tb = type builder, classqname = nom qualifié de la classe,
   fd = description d'un champ dans le langage IL *)

and ilfield tp classqname fd =
  let fname = fd.fnme in
  let fattr = List.map fieldattribute fd.fatt in
  (* définition du constructeur de champ *)
  let fieldbuilder = tp#define_field fname (iltype fd.fsig) fattr in
  Hashtbl.add transient_fields (classqname^"::"^fname) (fieldbuilder :> cFieldInfo)

(* tb = type builder, classqname = nom qualifié de la classe,
   m = description d'une méthode dans le langage IL,
   entrypoint permet de retourner le point d'entrée par effet de bord *)

and ilmethod entrypoint tb classqname m =
  let callconv = if List.mem MAstatic m.matt then RCCStandard else RCCHasThis in
  match m.mnme with
    | ".ctor" | ".cctor" ->
       let ctorbuilder = (* définition du constructeur de constructeur *)
           tb#define_constructor (List.map methodattribute m.matt) callconv (signature m.mprm) in
       (* ajout à la table des types définis *)
       Hashtbl.add transient_constructors (classqname^"::"^m.mnme) (ctorbuilder :> cConstructorInfo);
       (* définition du générateur de code CIL *)
       let ilgenerator = ctorbuilder#get_il_generator() in
         (* fonction effectuant la suite des opérations (sera appelée plus tard) *)
         (fun () -> List.iter (fun (t,_) -> ignore (ilgenerator#declare_local (iltype t))) m.locals;
                    ilcode ilgenerator (List.rev m.minst))
    | _ ->
       (* définition du constructeur de méthode *)
       let methodbuilder = tb#define_method m.mnme (List.map methodattribute m.matt)
                                            callconv (iltype m.mtyp) (signature m.mprm) in
       (* ajout à la table des types définis *)
       Hashtbl.add transient_methods (classqname ^ "::" ^ m.mnme) (methodbuilder :> cMethodInfo);
       (* si point d'entrée, le signifier par effet de bord *)
       if m.entrypoint then entrypoint := Some (methodbuilder :> cMethodInfo);
       (* définition du générateur de code CIL *)
       let ilgenerator = methodbuilder#get_il_generator() in
         (* fonction effectuant la suite des opérations (sera appelée plus tard) *)
         (fun () -> List.iter (fun (t,_) -> ignore (ilgenerator#declare_local (iltype t))) m.locals;
                    ilcode ilgenerator (List.rev m.minst))
```

Une fois instancié, et avant qu'il ne soit complet, chaque objet à émettre est placé dans une table de hachage correspondant à sa catégorie : `transient_types`, `transient_fields` ainsi que `transient_methods` et `transient_constructors`. La fonction `iltype`, utilisée dans le code ci-dessus mais non reproduite ici, permet de retrouver dans la table `transient_types` les classes en cours d'émission (ou alors de récupérer un descripteur au moyen de méthodes de l'API `Reflection` dans le cas de classes externes définis dans des assemblages dépendants), ce qui est nécessaire pour engendrer les références de types relatives à ces classes.

Le code ci-dessus fait également appel à des convertisseurs d'attributs, passant de la représentation interne du langage `IL` de OCamIL à ceux interfacés par OJacaré.NET. En voici un exemple simple :

```
let fieldattribute = function
    FAassembly -> RFAAssembly
  | FAinitonly -> RFAInitonly
  | FAprivate  -> RFAPrivate
  | FApublic   -> RFAPublic
  | FAstatic   -> RFAStatic
```

Pour le reste, on voit dans les codes Caml des fonctions `ilclass`, `ilfield` et `ilmethod` la façon dont est implanté le mécanisme de construction paresseuse des éléments constitutifs de l'assemblage.

Génération du code des méthodes. Reste la génération du corps des méthodes et des constructeurs, obtenue en utilisant la classe `ILGenerator` :

```
// fichier reflection.idl
// émission de code CIL

class ILGenerator {
  [name emit]void Emit(OpCode);
  [name emit_byte]void Emit(OpCode, byte);
  [name emit_short]void Emit(OpCode, short);
  [name emit_int]void Emit(OpCode, int);
  [name emit_long]void Emit(OpCode, long);
  [name emit_double]void Emit(OpCode, double);
  [name emit_typed]void Emit(OpCode, System.Type);
  [name emit_cstr]void Emit(OpCode, System.Reflection.ConstructorInfo);
  [name emit_meth]void Emit(OpCode, System.Reflection.MethodInfo);
  [name emit_fld]void Emit(OpCode, System.Reflection.FieldInfo);
  [name emit_str]void Emit(OpCode, string);
  [name emit_label]void Emit(OpCode, Label);
  [name emit_label_array]void Emit(OpCode, Label[]);
  [name define_label] Label DefineLabel();
  [name mark_label] void MarkLabel(Label);
  [name begin_exception_block] Label BeginExceptionBlock();
  [name begin_catch_block] void BeginCatchBlock(System.Type);
  [name end_exception_block] void EndExceptionBlock();
  [name declare_local] LocalBuilder DeclareLocal(System.Type);
}
```

Les méthodes `Emit` regroupent la génération de toutes les instructions CIL possibles, les différentes variantes ne différant que par le type des arguments nécessaires pour compléter une instruction : par exemple une instruction `ldc.i4` attend un entier, alors que l'instruction `call` a besoin d'un descripteur de méthode. On trouve aussi les méthodes permettant de gérer des blocs d'instructions et des étiquettes de code (pour les sauts).

Les instructions sont données par des instances de la classe `Opcode`, définis par des champs statiques de la classe `OpCodes` :

```
// fichier reflection.idl
// définition des instructions CIL

class OpCode {}

class OpCodes {
  [name opcodes__add] static OpCode Add;
  [name opcodes__and] static OpCode And;
  [name opcodes__box] static OpCode Box;
  [name opcodes__beq] static OpCode Beq;
  [name opcodes__bge] static OpCode Bge;
  [name opcodes__bgt] static OpCode Bgt;
```

```
  [name opcodes__ble] static OpCode Ble;
  [name opcodes__blt] static OpCode Blt;
  [name opcodes__br] static OpCode Br;
  [name opcodes__brtrue] static OpCode Brtrue;
  [name opcodes__brfalse] static OpCode Brfalse;
  [name opcodes__call] static OpCode Call;
  [name opcodes__callvirt] static OpCode Callvirt;
  [name opcodes__newobj] static OpCode Newobj;
  [name opcodes__convi] static OpCode Conv_I;
  [name opcodes__convi4] static OpCode Conv_I4;
  [name opcodes__convi8] static OpCode Conv_I8;
  [name opcodes__castclass] static OpCode Castclass;
...
// et ainsi de suite
}
```

Le rôle de la fonction Caml `ilcode` référencée dans `ilmethod` reproduite plus haut n'est plus alors que de convertir la suite d'instructions de la représentation interne `IL` en une suite d'appels aux fonctions `opcodes_*` correspondantes...

5.2.2.2 Application au toplevel

Une application directe du travail précédent est la compilation et l'exécution des phrases du toplevel OCamIL directement en mémoire, sans passer par le disque.

Cela présente plusieurs intérêts immédiats :

- Une meilleure intégration du toplevel, qui n'utilise pas de commande externe.
- Un fonctionnement moins exigeant en termes de droits de sécurité : une application qui n'utilise pas le disque local peut être plus facilement déployée, y compris à partir du Web (voir la section 5.1.2 sur les appliquettes).
- Une génération de code plus légère : l'API `Reflection` permet la génération incrémentale d'un assemblage en mémoire (ce qui n'est pas possible sur le disque) et autorise l'exécution de méthodes dans des classes engendrées faisant partie d'un assemblage non achevé. Cela permet donc de générer les instructions successives du toplevel dans un même assemblage au lieu de faire correspondre à chaque phrase Caml un assemblage différent. L'édition de liens s'en trouve d'autant facilitée.
- De meilleurs performances : on évite les latences dues aux accès disque, on fait l'économie des vérifications du chargeur de classes et de plus la structuration utilisant un seul assemblage est plus légère et plus performante.

Modifications de la génération de code. Il n'y a pas beaucoup de changements à effectuer pour permettre la génération incrémentale de code en mémoire, les constructeurs de l'API `Reflection.Emit` étant prévus pour cela. La fonction Caml `ilcompunit` doit simplement s'abstenir de créer un nouvel assemblage à chaque appel, ce que l'on fait en gardant dans des variables globales une instance des objets `AssemblyBuilder` et `ModuleBuilder` créés lors du premier appel. L'objet

`AssemblyBuilder` est également réutilisé pour effectuer les appels dynamiques de fonctions dans les types générés pour chaque phrase du toplevel.

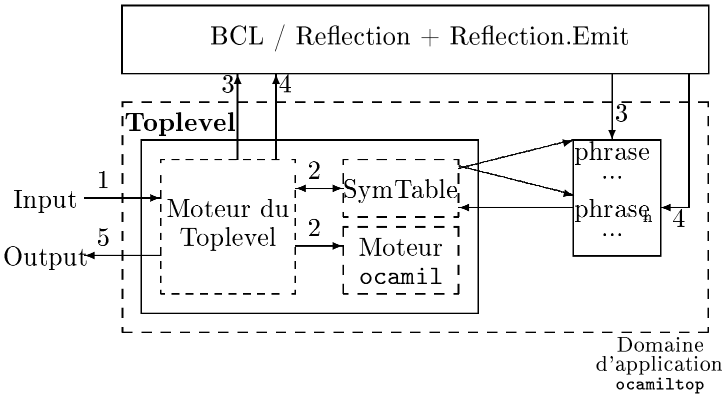


FIG. 5.13 – *Toplevel utilisant* `Reflection.Emit`

L'édition de liens pour chaque phrase du toplevel est plus simple puisqu'il n'y a plus qu'un seul assemblage généré. En particulier, les classes utilisateur générées pour les types algébriques se trouvent toutes dans le même assemblage et il est donc très facile de les référencer.

Le toplevel en ligne. Le toplevel est compilé comme un composant .NET indépendant (qui en particulier ne repose sur aucune commande externe). Il ne reste plus qu'à l'affranchir complètement du système de fichiers en supprimant les accès en lecture effectués pour l'ouverture des fichiers `.cmi` et `.cmx` contenant la description des fonctions de la bibliothèque standard.

Nous avons mis en place une version du toplevel sous forme d'appliquette en partant de l'interface graphique décrite à la section 5.2.1.2. Cette version surcharge à la fois les entrées-sorties standard et les accès en lecture au système de fichiers, en utilisant les techniques décrites à la section 5.1.1.1 :
- Les entrées et sorties standard sont interfacées avec des flux personnalisés, reliés à des composants graphiques de saisie et d'affichage (identiques au toplevel graphique.)
- Les flux de fichiers en lecture sont initialisés à partir de fichiers situés sur le serveur (on repose sur les classes `HttpWebRequest` et `HttpWebResponse`). Cela permet de diriger les requêtes de fichiers `cmi`/`cmx` de la librairie standard vers le serveur Web.

Au total la couche additionnelle écrite pour le déploiement sous forme d'appliquette consiste en à peine 200 lignes de code C#. La figure 5.14 donne un aperçu du résultat.

Alternativement, il est aussi possible de compiler le code du toplevel en incluant les fichiers `.cmi` et `.cmx` dépendants sous forme de ressources, et de surcharger les

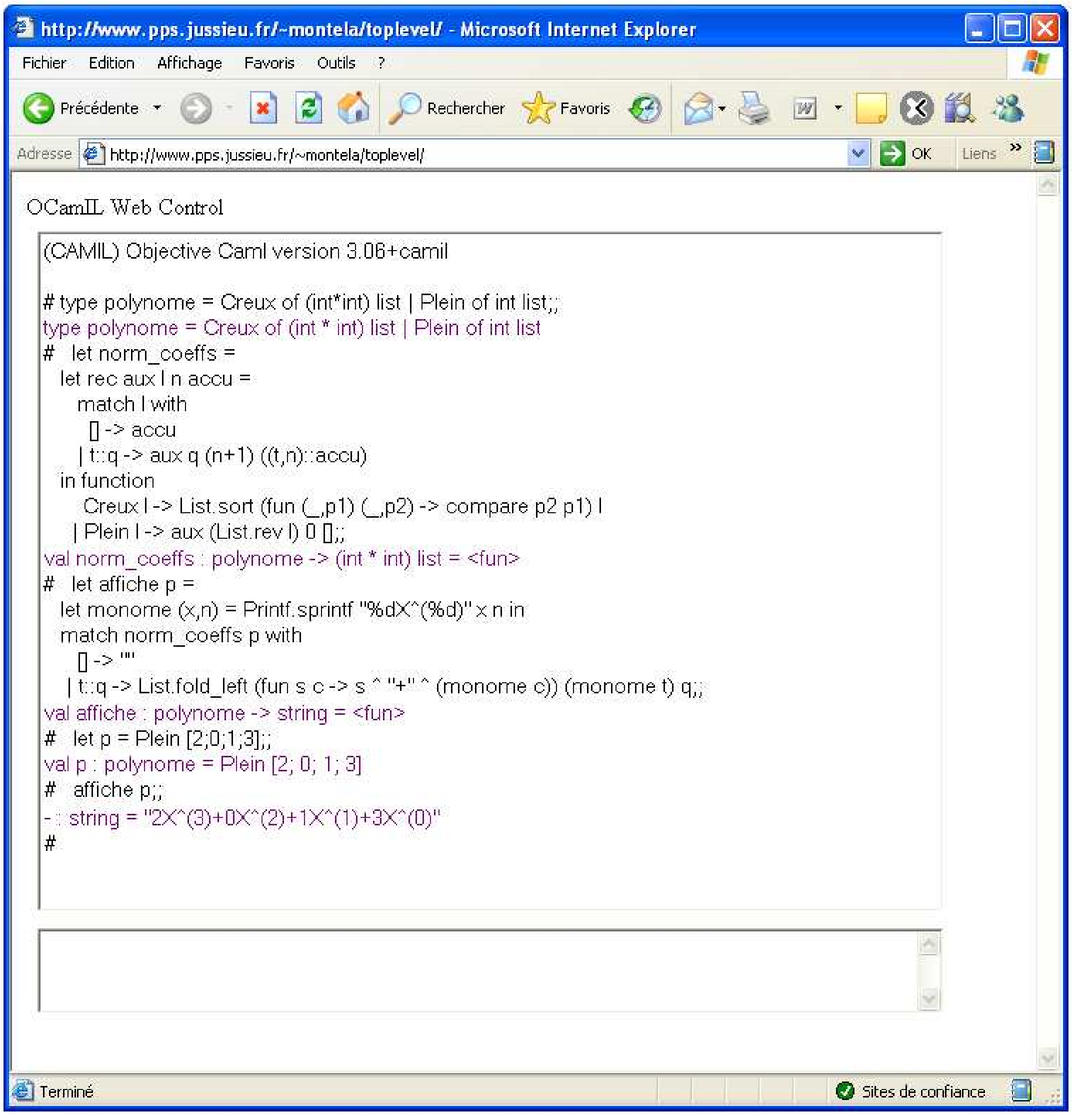


FIG. 5.14 – *Le toplevel comme appliquette*

flux de fichiers en lecture de manière à ce qu'ils se branchent sur ces ressources embarquées. Le composant toplevel forme dans ce cas une unité totalement autonome !

Embarquer le toplevel Caml sous forme de composant .NET a des applications dépassant le cadre de Caml. Le langage SCOL [8], inspiré de Caml et écrit en Caml, pourrait bénéficier de cette technologie. De plus, l'adaptation[2] de CamlP4 [31] permettrait de faire rentrer d'autres langages dans l'univers .NET.

2. Malheureusement, l'implantation de CamlP4 a recours à la fonction `Obj.magic` pour contourner le système de types de Caml, d'une manière incompatible avec OCamIL.

5.3 Sérialisation sûre

La sérialisation d'une valeur consiste à la représenter sous la forme d'une suite d'octets qui pourra être sauvegardée dans un fichier dans le but de la relire ultérieurement ou de la communiquer à d'autres programmes. Il est cohérent que les langages statiquement typés cherchent à offrir lors de la manipulation des valeurs dé-sérialisées les même garanties de sûreté que pour les autres valeurs. Cela nécessite toutefois de joindre à la représentation des valeurs sérialisées une information permettant de leur attribuer un type correct lors de la dé-sérialisation. L'implantation standard de Objective Caml ne le fait pas et présente donc un défaut de sûreté dans ce domaine.

D'un autre côté la plate-forme .NET maintient les types exacts (classes de construction) des valeurs à l'exécution et propose un mécanisme de sérialisation qui sauvegarde ces types avec les valeurs. Il en résulte que les programmes Objective Caml compilés avec OCamIL bénéficient directement de la sérialisation sûre de .NET. Nous en détaillons les propriétés et les limites dans la suite.

5.3.1 Caractéristiques de la sérialisation sous .NET

Les types standard (types de base, tableaux...) du CTS sont automatiquement sérialisables. Les classes du CTS qui sont décorées par l'attribut `serializable` bénéficient de l'API de sérialisation de la plate-forme (c'est aussi le cas des types valeurs structurés). La sérialisation sous .NET gère les partages, si bien qu'il est possible de sérialiser des valeurs circulaires en temps fini.

5.3.1.1 Sûreté

La sûreté vient du fait que la sérialisation .NET enregistre les méta-données (en particulier les types) avec les valeurs. Dé-sérialiser une valeur ne provoque normalement jamais d'erreur (sauf en cas de défaut de la source, comme un fichier corrompu ou une connexion réseau coupée ; mais il est de toutes façons possible de récupérer ces erreurs) et retourne une valeur de type `object`. C'est lors d'un transtypage de ce résultat vers le type attendu que peut se produire une erreur de types et donc la levée d'une exception `System.InvalidCastException`. Il n'y a jamais d'interruption brutale du programme et l'exception est toujours rattrapable. Nous pouvons exploiter cette caractéristique sous OCamIL.

5.3.1.2 Restrictions

Pour qu'une valeur soit reconnue être du bon type et passer le test du transtypage, il est nécessaire que le type utilisé à la dé-sérialisation soit un sur-type du type exact de la valeur sérialisée, c'est-à-dire de sa classe de création.

Rappelons que la relation de sous-typage dans le CTS ne s'appuie pas sur la structure des types mais sur une hiérarchie de classes et d'interfaces nommées, elles-mêmes subordonnées à une déclaration dans un espace de noms et un assemblage donnés. Ainsi deux classes structurellement équivalentes mais de noms différents, ou même structurellement équivalentes et de noms identiques mais déclarées dans des assemblages différents, seront incompatibles.

Cela impose la restriction suivante : les types impliqués dans une sérialisation / dé-sérialisation doivent être déclarés dans un assemblage partagé par les programmes respectivement responsables de la sérialisation et de la désérialisation (ce qui est automatiquement le cas quand ces deux programmes sont identiques).

Dans le cas de OCamIL, étant donné le découpage entre modules et assemblages, cela signifie qu'il ne suffit pas que le ou les modules définissant des types utilisateurs voués à être sérialisés soient partagés par les différents programmes ; il faut aussi qu'ils soient tous compilés dans une bibliothèque unique qui sera liée à ces programmes.

5.3.2 Application à Objective Caml

L'application directe de la sérialisation .NET à OCamIL apporte un mécanisme toujours sûr, mais il reste des imperfections au niveau de l'intégration avec le système de types Caml.

5.3.2.1 Valeurs sérialisables

Tous les types CTS manipulés par OCamIL (types de bases, structures génériques comme les tableaux d'objets, mais aussi les classes engendrées) sont automatiquement compilés avec l'attribut `serializable`. Les fonctions `Marshal.from_channel` et `Marshal.to_channel` de la librairie standard sont branchées sur l'API de sérialisation de la plate-forme .NET.

Les types de valeurs Caml suivants sont sérialisables sous .NET avec OCamIL :
- les types de base,
- les structures génériques (tableaux, n-uplets, variants polymorphes),
- les types algébriques définis (variants et enregistrements),
- les valeurs fonctionnelles (y compris les fermetures à environnement, récursives et mutuellement récursives),
- les objets.

La sérialisation des exceptions n'est pas exploitable avec leur représentation actuelle mais peut l'être en modifiant légèrement OCamIL[3].

5.3.2.2 Types CTS et types Caml dans la sérialisation

La sûreté de la sérialisation sous .NET s'exprime au niveau des représentations des valeurs dans le CTS. Sous OCamIL, toute tentative de dé-sérialiser une valeur

3. Les exceptions sont sérialisables mais ne sont actuellement pas utilisables après dé-sérialisation. La raison est que chaque exception est identifiée par une chaîne de caractères, et le test d'égalité de chaînes est ici une égalité de références et non de valeurs, ce qui permet à la fois de gérer simplement des exceptions homonymes déclarées dans des modules différents et d'avoir un test plus rapide. En conséquence le test d'égalité est toujours faux sur une exception dé-sérialisée.

ayant un type différent du type attendu (plus précisément, un type CTS différent) a pour conséquence la levée d'une exception `System.InvalidCastException`, qui est récupérable dans le code OCamIL (voir la section 4.2.2.1), comme dans l'exemple suivant :

```
type t = A | B of int

let export fname v =
  let oc = open_out_bin fname in
  Marshal.to_channel oc v [];
  close_out oc

let import fname =
  let ic = open_in_bin fname in
  let v = (Marshal.from_channel ic : t) in
  close_in ic;
  v

let _ = export ".save" (B 33)

(* Plus tard, lors d'une autre exécution *)

let _ =
  try
    let v = import ".save" in ...
  with CLIException("System.InvalidCastException",_) -> ...
```

La fonction `Marshal.from_channel : in_channel -> 'a` retourne toujours un objet sous OCamIL mais il se peut que cet objet n'ait pas le type exact attendu. Dans l'exemple précédent, le cast Caml explicite `(Marshal.from_channel ic : t)` permet d'engendrer dans le code compilé une instruction de cast CIL qui lève une exception `System.InvalidCastException` en cas de problème.

Le comportement d'une valeur Caml dé-sérialisée et le moment où l'exception `System.InvalidCastException` va être déclenchée dépend donc de l'adéquation entre types Caml et types CTS. Les deux modes de compilation de OCamIL (reconstruction et propagation des types) sont sûrs du point de vue de la sérialisation mais c'est la compilation OCamIL par propagation de types qui donne la meilleure discrimination (le recours à des types CTS spécialisés permet une détection plus rapide des erreurs de types en comparaison avec des représentations génériques). D'autre part les représentations génériques partagées entre types Caml d'implantations compatibles rend possible l'utilisation de valeurs sous un type différent de leur type de création, ce qui est sûr mais incorrect (on aimerait pouvoir le détecter).

La discussion qui suit est fondée sur les représentations choisies par OCamIL en mode propagation de types.

Les types de base. Ils sont bien compartimentés et ils déclenchent immédiatement une exception en cas de problème.

Les variants et les enregistrements. Ils sont représentés par des classes spécialisées et déclenchent immédiatement une exception au niveau du transtypage vers le type attendu en cas d'incompatibilité. Cependant il y a deux cas où la détection d'une incompatibilité est plus tardive :

- Pour les constructeurs constants d'un type variant : le compilateur OCamIL optimise la représentation des constructeurs constants par des valeurs prédéfinies de la classe `CamIL.Variant` qui sont partagées par des types variants différents. Il est alors possible de sérialiser un constructeur constant d'un type variant `t1` et de le dé-sérialiser comme une valeur de type `t2` si celui-ci déclare un constructeur constant de même tag. Il n'y a pas de risque de crash du programme mais on passe à côté d'une détection d'incompatibilité de types et donc d'une erreur potentielle. Pour éviter ce problème on peut les représenter par des valeurs prédéfinies de la classe abstraite correspondant à un type variant donné (voir la section 3.1.2.2).
- Les types variants et enregistrements paramétrés sont représentés par une même classe CTS générique. Par exemple sérialiser une liste de flottants et la dé-sérialiser comme une liste d'entiers ne provoque immédiatement une erreur. Il faudra attendre d'opérer sur les éléments de la liste pour qu'une exception soit lancée, signalant l'incompatibilité entre entiers et flottants, comme dans le programme suivant :

```
let export fname v =
  let oc = open_out_bin fname in
  Marshal.to_channel oc v [];
  close_out oc

let import fname =
  let ic = open_in_bin fname in
  let v = (Marshal.from_channel ic : float list) in
  close_in ic;
  v

let _ = export ".save" [1 ; 2 ; 3]

(* Plus tard, lors d'une autre exécution *)
let _ =
    let v = import ".save" in (* pas d'exception *)
    (* l'exception est levée lors de la division *)
    List.map (fun x -> x /. 2.) v
```

Dans l'exemple précédent, bien que la fonction `import` comporte un cast explicite, une liste d'entiers ne provoque pas la levée d'une exception car la classe OCamIL utilisée pour les listes est générique. L'exception est levée lors de la première division flottante car la valeur en tête, qui est ici un entier boxé caché derrière le type `object`, est dés-encaspulée et transtypée vers un flottant pour satisfaire l'opérateur de division flottante, transtypage qui est impossible.

Les n-uplets, tableaux et variants polymorphes. Ils sont tous représentés par des tableaux d'objets et il y a donc la possibilité de confusion de type non-détectable à la dé-sérialisation (on peut sérialiser un n-uplet et le dé-sérialiser comme un tableau par exemple).

Les fermetures. Elles héritent de la classe `CamIL.Closure` et sont donc isolées des autres types de valeurs, mais cependant rien dans ce type générique n'indique le type de la fonction lui-même ; on peut donc avoir des problèmes lors de l'application d'une fermeture dé-sérialisée. La capture de l'exception `System.InvalidCastException` devra alors être faite au moment de l'application, comme dans l'exemple suivant :

```
let export fname v =
  let oc = open_out_bin fname in
  Marshal.to_channel oc v [];
  close_out oc

let import fname =
  let ic = open_in_bin fname in
  let v = (Marshal.from_channel ic : int -> int) in
  close_in ic;
  v

let _ = export ".save" string_of_int

(* Plus tard, lors d'une autre exécution *)
let _ =
    let f = import ".save" in (* pas d'exception *)
    (* l'exception est levée dans la suite *)
    let v = f 3 in (* v vaut la chaîne "3")
    v + 1
```

Dans l'exemple précédent la fonction `string_of_int` a pour type `int -> string`. La dé-sérialisation ne lève pas d'exception, même en présence du cast explicite vers `int -> int` dans la fonction `import` car à ce stade on peut seulement vérifier que la valeur dé-sérialisée a le type des fermetures `CamIL.Closure`, ce qui est bien le cas. L'application de la fonction renvoit la chaîne `"3"` et il y a levée d'une exception à deux endroits possibles : soit immédiatement lorsqu'une variable locale de type `int32` est allouée dans la méthode courante pour stocker la valeur de `v`, soit lors de l'addition du résultat à un entier (tout dépend des optimisations du compilateur).

Insistons sur le fait que le code d'une fermeture n'est pas sérialisé. Pour que la valeur dé-sérialisée fonctionne, il faut que la classe définissant la fermeture soit liable dynamiquement (c'est en particulier le cas quand la fermeture est construite dans un même programme effectuant sérialisation et dé-sérialisation).

Les objets. Ils ont une représentation générique à base de tableaux et donc leur sérialisation peut provoquer des confusions de types (ils peuvent être sérialisés les uns à la place des autres sans considération pour leur type structurel, ou être confondus avec des tableaux par exemple) et de plus le même problème que pour les fermetures se pose à l'application des méthodes. D'autre part le tableau des méthodes virtuelles est stocké dans un tableau d'objets complètement générique, ce qui rend possible l'appel d'une méthode qui n'existe pas : une exception est alors lancée.

La sérialisation couvre la plupart des valeurs Caml et elle est sûre, quitte à différer la détection d'une incompatibilité de types lors de l'utilisation effective de la valeur dé-sérialisée. Cependant il est tout de même souhaitable d'être fixé sur la validité d'une valeur dès sa dé-sérialisation. Pour ce faire il est possible d'effectuer des vérifications complémentaires sur la valeur dé-sérialisée en la visitant au moyen des méthodes d'introspection .NET (l'API `Reflection` interfacée avec OJacaré.NET si on doit le faire à la main).

Nous pouvons aussi nous s'inspirer des travaux présentés dans [47], qui introduisent une opération de dé-sérialisation sûre et un type `'a tyrepr` décrivant la grammaire de types Caml, servant à paramétrer la dé-sérialisation :

```
SafeUnmarshal.from_channel : 'a tyrepr -> in_channel -> 'a

let _ =
  try
    let ic = open_in_bin ".save" in
    let _ = SafeUnmarshal.from_channel [^ float list ^] ic in
    ...
  with Fail -> ...
```

Il serait envisageable de l'implanter en générant automatiquement une séquence d'appels à l'API `Reflection`, déterminée à partir de l'expression de types donnée en argument. Par exemple pour le type `float list`, on peut engendrer un parcours de la liste qui vérifie bien que les éléments sont des flottants. On peut parfois simplifier en exploitant le typage statique de Caml : dans le cas précédent il suffit de tester le premier élément de la liste.

Bien que les fermetures ne soient pas traitées dans [47], il doit être possible dans le cadre de OCamIL d'analyser les objets fermeture : leur analyse permet de connaître le nombre d'arguments déjà appliqués, et il reste ensuite à analyser la signature de la méthode `exec` (voir la section 3.1.3.1) à l'aide de l'API `Reflection`.

Chapitre 6

Tests de performance

Ce chapitre est consacré à des tests de performance. Il compare différentes approches possibles au sein du compilateur OCamIL et le situe par rapport aux compilateurs standard de l'INRIA ainsi que d'autres compilateurs de langages de la famille ML sur .NET.

Sommaire

6.1 Choix guidés par les performances de la plate-forme

6.1.1 Méthodologie

Nous avons réalisé des tests de bas-niveau en C# et utilisé les services de la plate-forme .NET en matière de mesure d'intervalles de temps (classes `DateTime` et `TimeSpan`), qui fournissent le temps réel total d'exécution (cela ne détaille pas les temps d'exécution des différents processus et du système). Les tests sont itérés un nombre de fois suffisant pour que l'exécution dure au moins 10 secondes. Les mesures sont précises à la milliseconde près, mais nous avons choisi de présenter des tests de performances relatives en normalisant à chaque fois l'un des candidats à 1 et en exprimant les écarts sans garder plus de trois chiffres significatifs. Les valeurs données dans les tableaux reflètent les temps d'exécution si bien que plus une valeur est faible, meilleure est la performance.

Il faut noter qu'en raison de l'action du ramasse-miettes, que l'on ne peut pas contrôler, les tests de cette section ne rendent pas compte du coût exact d'une opération isolée mais plutôt de cette opération dans un contexte réel.

6.1.2 Structures de données

6.1.2.1 Représentations des types algébriques

Nous évaluons dans cette section des représentations concurrentes pour les types algébriques de Caml, à savoir les enregistrements et les variants.

Enregistrements. Les enregistrements posent la question de l'efficacité des structures typées (classes dont les champs reçoivent le type le plus approprié) face aux structures génériques (les tableaux d'objets).

Les tests présentés sur le tableau suivant comparent la performance des opérations de *création*, d'accès en *lecture*, d'accès en *écriture* et de *copie* sur les représentations envisagées, pour un « panier » d'enregistrements possédant 2, 8 et 16 champs, en distinguant trois occupations possibles de ces champs : sur la ligne (R) uniquement des types références (objets, chaînes de caractères...), sur la ligne (V) uniquement des types valeurs (entiers, flottants...) et sur la ligne (M) un mélange équitable des deux. Pour chaque colonne les valeurs ont été rapportées à la plus petite d'entre elles.

	Création		Lecture		Écriture		Copie	
	classes	object[]	classes	object[]	classes	object[]	classes	object[]
(R)	1,3	2,8	1,0	20	1,0	64	1,5	6,4
(M)	1,2	3,7	1,3	19	13	120	1,2	6,7
(V)	1,0	4,9	1,3	26	3,7	130	1,0	6,6

Ces résultats confirment l'avantage des structures typées sur les structures génériques, ce qui justifie les efforts de développement en direction de la propagation des types dans le compilateur OCamIL.

Les structures génériques sous forme de tableaux d'objets coûtent en raison des opérations de (dés)encapsulation mais aussi à cause des tests de bornes. Certains chiffres sont toutefois assez surprenants : par exemple nous nous attendions à ce que les valeurs de la ligne (M) soient toujours entre les valeurs des autres lignes, de même que le gain d'utilisation des structures typées soit d'autant plus grand que la proportion de types valeurs augmente. Or ce n'est pas toujours le cas, certainement en raison de l'action du ramasse-miettes sur les valeurs des types références qui deviennent caduques au cours des tests et qui sont donc collectées. Ceci dit, les structures typées et les structures génériques sont testées dans les mêmes conditions et il est surprenant de voir le gain obtenu par ces dernières, en particulier dans la ligne (M). Les effets du ramasse-miettes doivent moins se faire sentir lors de tests sur des programmes réels (voir section 6.2.2.2).

Variants. Les représentations des types variants sont simulées sur la base de celles des enregistrements. Il faut ajouter la discrimination par un *tag*. Les deux structures testées sont les suivantes :
- des tableaux d'objets gardant le tag dans une case additionnelle,
- des classes aux champs typés de manière exacte, héritant d'une classe mère `Variant` qui définit un champ tag.

Les opérations de création, lecture et copie sont identiques à celles des enregistrements vus précédemment. Nous reproduisons ici les résultats obtenus pour des opérations de filtrage sur les variants (test du tag suivi d'un downcast si nécessaire) en reportant également les valeurs obtenues ci-dessus pour les opérations de création et de lecture.

	Création		Lecture		Filtrage	
	classes	object[]	classes	object[]	classes	object[]
(R)	1,3	2,8	1,0	20	1,3	1,0
(M)	1,2	3,7	1,3	19	1,3	1,0
(V)	1,0	4,9	1,3	26	1,3	1,0

Les valeurs pour le filtrage montrent deux choses :
- les résultats ne dépendent pas de la nature des champs, qui ne sont pas sollicités lors de l'opération de filtrage elle-même,
- le filtrage des structures génériques est un petit peu plus efficace, ce qui montre que l'extraction d'un tag sous forme boxée est moins coûteuse que la lecture directe du champ entier dans la classe mère `Variant` suivie d'un downcast.

Le test de filtrage ci-dessus n'est pas réaliste car il simule un filtrage sans lecture des données contenues dans les variants. Les véritables performances sont situées entre celles du filtrage et de la lecture (les deux tests faisant tourner un nombre comparable d'opérations) et nous estimons donc que les structures génériques sont moins performantes en situation réelle, faisant intervenir quelques opérations de lecture en moyenne. Les tests de la section 6.2.2.2 sont effectués sur de véritables programmes Caml, et le programme `Morematch` éprouve spécifiquement le filtrage de motifs.

6.1.2.2 Représentation des chaînes de caractères

Comme indiqué à la section 3.1.1.1, les trois choix les plus raisonnables pour la représentation des chaînes de caractères sont les types `string`, `StringBuilder` et `char[]` (à noter cependant que le type `string` ne permet pas de représenter des chaînes mutables).

Nous avons testé les performances de ces trois représentations pour différents types d'opérations élémentaires, à savoir : *l'instanciation dynamique* (c'est-à-dire l'allocation d'une chaîne de caractères d'une taille donnée), *l'instanciation par littéral* (la création d'une instance de la chaîne à partir d'une constante écrite en dur dans le code), *la copie*, le calcul de *la longueur*, *la lecture* et *l'écriture* d'un caractère dans la chaîne, *la concaténation* et *le remplacement* des occurrences d'un caractère. Notons que ces deux dernières opérations ont été envisagées sous un angle fonctionnel, c'est-à-dire n'ayant pas lieu en place, mais donnant leur résultat sous forme d'une nouvelle chaîne. Chaque opération a été réalisée entre un million et un milliard de fois, pour 4 tailles de chaînes différentes : les tailles 1, 10, 100 et 1000. Le tableau suivant présente le temps mesuré, relativement à la représentation `char[]` qui a été normalisée à un (celle-ci donnant des résultats intermédiaires pour la majorité des tests). Pour chaque opération, quand plusieurs implantations s'offraient à nous pour une représentation donnée, nous avons retenu la plus rapide.

	`StringBuilder`	`char[]`	`string`
Instanciation (dynamique)	4,1	1,0	3,1
Instanciation (par littéral)	1,1	1,0	0,00023
Copie	1,1	1,0	0,73
Longueur	0,76	1,0	0,66
Accès en lecture	1,0	1,0	1,0
Accès en écriture	11	1,0	
Concaténation	1,6	1,0	0,76
Remplacement	1,3	1,0	0,97

Ces tests montrent que pour les chaînes non mutables, la représentation `string` est en effet la meilleure. L'instanciation d'un littéral a d'ailleurs un coût négligeable (car ils sont stockés directement dans les méta-données des fichiers PE sous leur forme définitive). L'instanciation dynamique est assez coûteuse en l'absence d'opération dédiée, mais ce n'est de toutes façons pas une opération courante (allocation de tampons par exemple).

En ce qui concerne les chaînes mutables, c'est la représentation `char[]` qui sort vainqueur : elle est presque toujours mieux classée que `StringBuilder` (sauf pour la détermination de la longueur) et ses performances ne sont pas si éloignées de celles du type `string`. Alors que les temps de lecture sont semblables pour les trois représentations, l'écriture est bien plus rapide avec `char[]`. Les opérations de concaténation et de remplacement, qui combinent les opérations précédentes, donnent logiquement le même tiercé.

Du fait que la classe `StringBuilder` encapsule une instance de la classe `string`, les résultats précédents ne sont pas surprenants. Le compilateur OCamIL choisit la représentation `char[]` pour les chaînes de caractères, qu'il veut garder mutables. Une optimisation possible (mais non implantée) consisterait à utiliser localement le type `string` en utilisant un algorithme d'analyse de code : des valeurs ne pouvant pas sortir d'un module dans lequel elles ne sont jamais utilisées en écriture peuvent être représentées par le type `string`.

En raison de la normalisation des résultats, le tableau ci-dessus ne permet pas de comparer les coûts respectifs des différentes opérations. Cela dit nous n'avons pas trouvé de manière convaincante d'estimer dans quelles proportions peuvent être utilisées ces différentes opérations dans un programme « typique ». La section 6.2.2.2 compare les performances des différentes représentations dans le cadre de programmes compilés par OCamIL (moyennant une bascule en ligne de commande permettant de changer de représentation) et permet de se faire une idée sur quelques cas concrets.

6.1.3 Structures de contrôle

6.1.3.1 Représentation des fermetures

La section 3.1.3.1 a décrit la représentation des fermetures dans le compilateur OCamIL (qui utilise une classe pour chaque fermeture afin d'implanter l'environnement au moyen de champs les plus précis possibles). L'article [11] compare les performances de cette solution avec celles d'une classe unique dont l'environnement est générique, pour lequel le dispatch vers le code de chaque fonction est codé module par module et repose sur un identificateur entier contenu dans chaque instance de fermeture. Dans le cas d'un langage typé dynamiquement comme Scheme cette deuxième solution s'avère plus efficace. Nous effectuons des tests analogues dans le cadre d'un langage typé statiquement comme Objective Caml.

Nous testons la création et l'application générique (c'est-à-dire sans connaître localement l'identité de la fermeture) de plusieurs fermetures à environnement. Deux types d'implantation sont évalués : à base de classes spécialisées ou avec une classe générique implantant le mécanisme de dispatch (avec dans ce cas deux variantes pour cette classe, l'une représentant l'environnement comme un tableau d'objets et l'autre comme des champs de type `object`) :

- Les classes spécialisées sont décrites à la section 3.1.3.1. Les fermetures sont représentées par des classes héritant d'une classe mère qui déclare une méthode d'application générique. Les différentes classes implantent leur environnement de manière spécialisée au moyen de champs du bon nombre et des bons types. C'est la liaison tardive qui dirige l'application générique vers le code de la fonction.

– L'implantation à base de dispatch « ah-hoc » n'utilise qu'une seule classe. Celle-ci définit un identificateur de fermeture représenté par un champ entier, et un environnement générique (selon les deux variantes considérées : soit comme un tableau d'objets, soit comme un nombre borné de champs de type `object`, la borne étant déterminée statiquement pour un module donné). La classe contient une méthode d'implantation pour chaque fonction définie dans le module et la méthode d'application générique se base sur l'identificateur entier pour dispatcher l'appel vers le code correspondant.

Nous testons à la fois pour des environnements composés majoritairement de types références ou de types valeurs. Les résultats sont normalisés sur la colonne des classes spécialisées.

Création	Classe générique (tableau)	Classe générique (champs)	Classes spécialisées
Types références	3,7	1,2	1,0
Types valeurs	4,9	2,2	1,0
Application	Classe générique (tableau)	Classe générique (champs)	Classes spécialisées
Types références	1,0	0,99	1,0
Types valeurs	1,1	1,04	1,0

Pour les tailles typiques des environnements de fermetures (moins de 20 champs), l'allocation de tableau pénalise systématiquement les classes génériques par tableau. Les performances des classes génériques par champs sont très bonnes, quasi-identiques à celles des classes spécialisées pour des environnements composés de types références, mais perdent du terrain pour les environnements composés de types valeurs.

Bien que les écarts soient plus nets que pour la création des fermetures que pour leur application, ces mesures confortent notre choix d'utiliser des classes spécialisées dans OCamIL.

6.1.3.2 Performance des exceptions

Nous comparons deux implantations des exceptions : la première utilise les exceptions de la machine .NET et la deuxième consiste à gérer les exceptions à la main comme suggéré à la section 3.1.3.3, par l'encapsulation des valeurs de retour des fonctions dans un objet pouvant contenir soit une valeur soit une exception.

Détaillons la gestion manuelle des exceptions : celle-ci distingue les manières de les propager à l'intérieur et à l'extérieur des méthodes. Dans une méthode, les blocs de traitement des exceptions sont étiquettés et sont rejoints par des instructions de branchement. Une variable locale de la méthode est chargée de relayer l'exception à transmettre. Pour les récupérateurs d'exceptions imbriqués, du code généré se charge de transmettre les exceptions non rattrapées au niveau supérieur.

Du fait que Objective Caml suit une convention d'appel par valeurs, les exceptions déclenchées dynamiquement par l'évaluation de sous-expressions sont obligatoirement issues d'une application de fonction. Le retour des fonctions est de deux sortes : valeur ou exceptions. Chaque appel de fonction est suivi d'une analyse de son retour, avec propagation d'une exception ou affectation de la valeur de retour, suivant les cas.

Un exemple simple permet de donner une idée du code de manipulation des exceptions dans les deux cas. On suppose donnée une fonction `g` des entiers dans les flottants, pouvant lever une exception `Failure` (entre autres). Le code d'implantation d'une fonction `f` cliente de `g` est comme suit :

Code Caml
`let f x = try int_of_float (g x) with Failure -> 0`

Avec exceptions .NET

```
int f (int x) {
 .locals(int)
 .try {
   ldarg.0
   call double g(int)
   call int int_of_float(double)
   stloc.0
   leave A:
   }
 catch (Exception) {
   isinst Failure
   brfalse P:
   ldc.i4.0
   stloc.0
   leave A:
P: rethrow
   }
A: ldloc.0
   ret
}
```

Avec gestion manuelle

```
IntEx f (int x) {
 .locals (Exception,DoubleEx)
   ldarg.0
   call DoubleEx g(int)
   dup
   stloc.1
   ldfld Exception DoubleEx::exn
   dup
   stloc.0
   brtrue H:
   ldloc.1
   ldfld double DoubleEx::value
   call int int_of_float(double)
   newobj void IntEx::.ctor(int)
   ret
H: ldloc.0
   isinst Failure
   brfalse P:
   ldc.i4.0
   newobj void IntEx::.ctor(int)
   ret
P: ldloc.0
   newobj void IntEx::.ctor(Exception)
   ret
}
```

Dans le deuxième cas on utilise des enveloppes valeurs/exceptions pour chaque type de base et au delà (le champ `value` contient la valeur si elle existe et le champ `exn` l'exception ; on teste si ce dernier champ est `null` pour distinguer les deux cas). On voit qu'il est nécessaire d'adapter les prototypes des fonctions. Par ailleurs dans les deux cas il est possible d'embarquer des paramètres dans l'objet exception, mais ce n'est pas illustré dans l'exemple ci-dessus.

Les tests de cette section consistent à lever une exception dans des appels de fonction et à rattraper cette exception à des niveaux de profondeur de pile variables (ici 5, 15, 30 et 100).

Les deux premières colonnes correspondent à des représentations utilisant les exceptions de la plate-forme .NET ; la différence se situant au niveau des paramètres des exceptions, représentés soit comme des structures génériques à base de tableaux d'objets, soit à base de classes aux champs spécialisés (une distinction semblable à celle analysée pour les types algébriques à la section 6.1.2.1). La troisième colonne est destinée aux exceptions gérées manuellement. Les résultats sont relatifs au test le plus performant, obtenu par la troisième implantation sur une profondeur de 5.

	Exceptions structurées	Exceptions génériques	Gestion manuelle
Profondeur 5	120	120	1,0
Profondeur 15	180	180	1,7
Profondeur 30	260	270	2,7
Profondeur 100	640	640	6,9

Ces tests révèlent la lourdeur des exceptions natives de la plate-forme. La différence entre les représentations par structures génériques et spécialisées est en faveur des structures spécialisées, mais d'un facteur inférieur à 1% si bien qu'il n'apparaît pas dans les résultats. Cette différence de vitesses est masquée par l'inertie du système d'exceptions lui-même. Le faible écart entre les deux choix a conduit le compilateur OCamIL à se contenter d'une représentation générique.

Les performances mesurées ici pour la gestion manuelle confirme l'analyse de l'article [51]. Cependant nous n'adoptons pas cette alternative pour plusieurs raisons : elle ne permet pas une bonne intégration du code produit par OCamIL avec des composants provenant d'autres sources dans l'optique de transmettre des exceptions d'un langage à un autre, et alourdit considérablement les appels de fonctions, les rendant moins efficaces et moins lisibles (ce qui pose problème pour le débogage). De plus dans un cadre de compilation modulaire, il peut être difficile d'analyser statiquement les sites susceptibles de transmettre des exceptions afin de minimiser l'insertion du code enveloppant les appels de fonctions : en effet une exception peut être levée dans un composant échappant à l'analyse statique, comme un composant issu d'un autre langage. On peut se référer à [62] pour une étude statique des exceptions non rattrapées en Caml.

Il faut retenir que les exceptions .NET sont conçues pour être utilisées de manière exceptionnelle, ce qui n'est pas malheureusement pas toujours le cas dans un bon nombre de programmes fonctionnels qui s'en servent comme un mécanisme de contrôle à part entière. Une optimisation peut consister à analyser les exceptions ne pouvant ex-filtrer d'un ensemble de fonctions dans un module afin de réaliser une implantation à base de branchements (ou d'enveloppes comme celles de la troisième méthode utilisée dans les tests ci-dessus). Les tests de la section 6.2.3 montrent l'impact des exceptions sur les performances de programmes Caml.

6.2 Performances du compilateur OCamIL

Nous testons dans la suite l'efficacité des exécutables produits par OCamIL et les confrontons à des versions issues d'autres compilateurs. Nous comparons également différentes options de compilation de OCamIL au regard des performances qu'elles induisent.

6.2.1 Méthodologie

Afin de pouvoir comparer les programmes compilés par OCamIL aux versions ne tournant pas sous .NET, il nous faut un moyen universel de mesurer le temps. Le système Windows ne propose pas l'équivalent de la commande Unix `time`, qui distingue les temps utilisateur et système d'un processus.

Nous nous rabattons sur la commande `time` tournant sous `cygwin` (`www.cygwin.org`, une couche Unix pour Windows). Même si on ne peut pas directement comparer les temps avec ceux obtenus par une autre méthode à la section 6.1, nous mesurons également les temps réels dans les tests suivants (cela intègre donc le temps système). Nous reproduisons plusieurs fois les tests afin de consigner des moyennes.

Tout au long de la section nous nous appuierons sur un jeu de tests, récapitulé sur la figure 6.1.

	style fonctionnel	variants	enregis-trements	poly-morphisme	exceptions	calcul flottant
`Arbl`	oui	oui		(listes)		
`Avl`	oui	oui	oui	oui	oui	
`Bdd`	oui	oui		(listes, tableaux)		
`Boyer`	oui	oui	oui		oui	
`Derive`	oui	oui		oui		
`DivEuclid`	oui	oui		oui		
`Fft`						oui
`Integrale`						oui
`KB`	oui	oui		oui	oui	
`Morematch`	oui (1)	oui	oui	oui		
`Morematch2`				oui		
`Nucleic`			oui			oui
`Pgcd`						
`Quicksort`				(tableaux)		
`Sieve`	oui			(listes)		
`SoliLet`				(tableaux)	oui	
`Syrac`						
`Tailcall`	oui (2)					
`TakE`					oui (3)	
`TakC`	oui					
`TakU`	oui					

(1) `Morematch` teste spécifiquement le filtrage de motifs
(2) `Tailcall` teste les appels récursifs terminaux
(3) `TakE` adopte un style de contrôle basé exclusivement sur les exceptions

FIG. 6.1 – *Les jeux de test et leurs spécificités*

Les tests `Pgcd`, `Sieve` (calcul de nombres premiers par la méthode du crible d'Eratosthène) et `Syrac` (vérification de la conjecture de Syracuse) font du calcul intensif sur les entiers et reposent sur la récursion terminale. `Tailcall` teste exclusivement cette dernière

De nombreux programmes testent des structures de données récursives (définies par des types variants) et le filtrage : `Derive` et `DivEuclid` font du calcul symbolique sur des termes, `Avl` travaille sur des arbres AVL, `Bdd` sur des arbres de décision binaires, `Arbl` sur des arbres lexicaux (ce test utilise de plus modules et foncteurs), et enfin `Boyer` et `KB` (algorithme de complétion de Knuth-Bendix) font du calcul intensif de termes ; le second utilisant de plus des exceptions à des fins de contrôle. Le filtrage de motifs est poussé dans ses derniers retranchements par la quarantaine de tests que rassemble `Morematch`. Le programme `Morematch2` effectue des tests similaires sur les variants polymorphes.

`TakE`, `TakC` et `TakU` sont trois versions d'un test utilisant la fonction de Takeuchi pour mettre à l'épreuve la rapidité d'appels de fonctions. `TakC` est une version curryfiée de `TakU` (ce dernier manipulant des triplets d'arguments) alors que `TakE` utilise des exceptions pour faire remonter des résultats de sous-appels.

Deux programmes testent les boucles : `Quicksort` (algorithme de tri) et `SoliLet` (résolution de puzzles).

Enfin, trois programmes testent le calcul de flottants : `Fft`, `Integrale` et `Nucleic`. Ce dernier est utilisé dans [45] pour comparer une douzaine de compilateurs de langages fonctionnels et utilise des structures de données basées sur des enregistrements.

La colonne « polymorphisme » indique les tests qui utilisent des types de données polymorphes, connus pour être coûteux dans l'implantation de OCamIL en absence de monomorphisation globale (voir la section 6.2.3). Les types de données polymorphes ne sont parfois simplement que des listes ou des tableaux.

La première colonne indique si les programmes réalisent de nombreux appels de fonctions.

6.2.2 Les différents réglages et leurs répercussions

6.2.2.1 Reconstruction et propagation de types

Nous comparons ici les performances des mêmes programmes, tantôt compilés avec l'option de reconstruction de types, tantôt avec l'option de propagation de types.

Comme nous l'attendions, les performances sont en majorité meilleures avec la propagation des types, aussi le tableau suivant donne-t-il les temps d'exécution des versions avec reconstruction relativement aux temps des versions avec propagation (un nombre supérieur à 1 signifie donc que la propagation de types est meilleure).

`Arbl`	`Avl`	`Bdd`	`Boyer`	`Derive`	`DivEuclid`	`KB`	`Nucleic`	`Quicksort`
0,95	1,26	2,11	1,06	1,46	2,04	0,92	1,56	1,44

Fft	Integrale	Pgcd	Sieve	SoliLet	TakE	TakU
0,96	1,04	0,99	1,00	0,97	0,97	1,01

Les programmes de tests sont regroupés en deux familles : sur la première ligne, ceux qui utilisent des types variants ou enregistrements et sur la seconde, ceux qui ne s'en servent pas.

La tendance qui se dégage est conforme à nos attentes : les programmes utilisant les types algébriques sont en général plus rapides, avec un facteur pouvant dépasser 2, alors que ceux qui ne les utilisent pas ont des performances quasi-identiques. La propagation de types permet l'utilisation de classes adaptées au mieux aux types algébriques.

Le cas de `KB`, qui utilise des types variants mais est moins efficace avec la propagation, illustre un problème lié à l'absence de monomorphisation. En effet ce programme a recours à des fonctions polymorphes pour manipuler toutes sortes de listes, et en particulier des listes d'association entre des entiers et des termes (définis par un type variant). Dans la version avec reconstruction de types, les représentations des valeurs sont, certes inefficaces, mais cohérentes entre elles (les cellules des listes sont des objets, les entiers sont encapsulés, y compris ceux décorant certains des termes) alors que dans la version avec propagation de types, certaines structures pas trop polymorphes sont typées finemenent (comme les termes) mais les listes sont toujours formées d'objets. Le test `KB` est justement un cas où de nombreux parcours et constructions de ces listes entraînent des opérations de conversions supplémentaires dans la version avec propagation, provoquant des performances inférieures.

Lorsqu'on adapte `KB` en monomorphisant manuellement les structures de données, on obtient des résultats meilleurs pour les deux versions, et dans cette nouvelle situation la « logique » est respectée : les performances sont meilleures quand on propage les types. La monophormisation des valeurs est certainement l'une des manières les plus efficaces pour améliorer les performances. L'utilisation des Generics de la plate-forme 2.0 et supérieures devraient être d'un grand secours en la matière.

Tout en offrant une meilleure visibilité des valeurs en phase de débogage, la propagation de types est profitable aux performances dans la majorité des cas. D'autre part elle apporte des propriétés de sûreté à la sérialisation des valeurs Caml (voir la section 5.3). Pour ces raisons, elle est retenue comme mode de fonctionnement par défaut du compilateur OCamIL.

6.2.2.2 Comparaisons de réglages fins

Représentation des types algébriques. En mode de propagation de types, le compilateur OCamIL permet de choisir de ne pas utiliser les classes dédiées pour les types algébriques, au moyen de bascules en ligne de commande. On peut ainsi forcer l'utilisation de structures génériques (tableaux d'objets) pour les enregistrements, les variants ou les deux ensemble.

Cela permet de comparer la qualité respective des types reconstruits et propagés pour tout le langage sauf les types algébriques. Le tableau suivant donne les temps d'exécution des mêmes tests utilisés à la section 6.2.2.1. Les valeurs obtenues pour la reconstruction de types et la propagation classique n'utilisant aucune structure générique sont reportées sur ce tableau et les résultats exprimés relativement à la propagation classique.

	Reconstr.	Propag.	Propag. E&V génériques	Propag. E génériques	Propag. V génériques
`Arbl`	0,95	1,00	1,05	1,00	1,00
`Avl`	1,26	1,00	1,26	1,11	1,14
`Bdd`	2,11	1,00	1,86	1,12	1,75
`Boyer`	1,06	1,00	1,07	1,00	1,07
`Derive`	1,46	1,00	1,52	1,00	1,50
`DivEuclid`	2,04	1,00	1,82	1,05	1,81
`KB`	0,92	1,00	1,01	1,00	1,01
`Nucleic`	1,56	1,00	1,59	1,45	1,12
`Quicksort`	1,44	1,00	1,08	1,06	1,00
`Fft`	0,96	1,00	1,02	1,06	1,00
`Integrale`	1,04	1,00	1,01	1,00	1,01
`Pgcd`	0,99	1,00	1,01	1,01	1,01
`Sieve`	1,00	1,00	1,08	1,00	1,08
`SoliLet`	0,97	1,00	1,03	1,03	1,03
`TakE`	0,97	1,00	1,00	1,00	1,00
`TakU`	1,01	1,00	1,00	1,00	1,00

Légende:
- E génériques = enregistrements génériques
- V génériques = variants génériques

Il est naturel de constater que les trois versions utilisant la propagation mais retombant sur au moins une sorte de structure générique sont systématiquement moins bonnes que la propagation utilisant des classes adaptées. De même l'utilisation d'enregistrements génériques seuls (respectivement de variants génériques seuls) est toujours plus efficace que l'utilisation simultanée d'enregistrements et de variants génériques. Les écarts sont d'autant plus sensibles que les programmes de test utilisent intensivement les types algébriques (tests de la moitié supérieure du tableau). Nos tests utilisent plus souvent des variants que des enregistrements ce qui explique leur plus gros impact sur les différences de performances.

Il est intéressant de comparer la première et la troisième colonne du tableau et de constater que les chiffres sont assez souvent à l'avantage de la reconstruction dans ce cas de figure. Nos expliquons cela de la manière suivante : propager les types et typer finement les valeurs dans les variables et les arguments de fonction mais pas pour les structures de données que l'on force artificiellement à être génériques crée un environnement hétérogène nécessitant davantage de transtypages (encapsulations et désencapsulations). La version avec propagation peut être plus efficace dans certaines circonstances où les types des variables et arguments de fonctions sont typés grossièrement par `object` et les valeurs passées par référence en retardant leur transtypage au moment de leur utilisation effective par une primitive.

Bien sûr ces mesures ne remettent pas en question la supériorité de la propagation de types et l'utilisation de structures de données adaptées aux enregistrements et aux variants.

Pré-allocation des constantes. Lorsqu'un type variant n'est constitué que de constructeurs constants, ses valeurs sont représentées par des entiers. Dans le cas contraire, ce sont des types référence qui sont utilisés dans tous les cas, y compris pour les constructeurs constants : des tableaux ne contenant que le tag (cas des structures génériques) ou des classes n'ayant pas d'autre champs que le tag (cas des structures fines autorisées par la propagation de types).

Toutefois les constructeurs constants ne contiennent pas d'autre information que leur tag et sont munis d'une sémantique par valeur. Afin de respecter cette sémantique et d'éviter l'allocation dynamique de constructeurs constants, ceux-ci sont alloués une seule fois à l'initialisation de l'environnement d'exécution OCamIL et ce sont ces représentants qui sont référencés dans tout contexte manipulant des constructeurs constants.

Le tableau suivant compare le comportement par défaut décrit ci-dessus à celui qui consiste à allouer dynamiquement des instances de constructeurs constants. Les valeurs sont les écarts relatifs de l'allocation dynamique par rapport à la pré-allocation.

`Arbl`	`Avl`	`Bdd`	`Boyer`	`Derive`	`DivEuclid`	`KB`	`Nucleic`	`Quicksort`
1,00	1,01	1,04	1,02	1,02	1,11	1,00	1,01	1,00

`Fft`	`Integrale`	`Pgcd`	`Sieve`	`SoliLet`	`TakE`	`TakU`
1,00	1,00	1,01	1,04	1,03	1,00	1,00

Les chiffres sont supérieurs à 1, ce qui signifie la pré-allocation, plus correcte au niveau de la sémantique (même si cela ne concerne que l'opération d'égalité physique `==`, rarement utilisée sur les variants, et n'a pas pas d'impact sur le filtrage de motifs), est également plus efficace.

Représentation des chaînes de caractères. Pour ces tests, nous exploitons une option du compilateur OCamIL permettant de changer de représentation de chaînes de caractères dans les exécutables produits. Il faut bien noter que tous ces tests ont été réalisés avec la même bibliothèque d'exécution OCamIL (à savoir l'assemblage `core_camil.dll`), ce qui signifie que toutes les primitives de manipulation de chaînes de caractères présentes dans cette bibliothèque sont les mêmes et ont la même signature. De toutes façons, ces primitives ont été choisies pour être les plus efficaces possibles indépendemment de la représentation choisie, et c'est le compilateur OCamIL qui se charge d'insérer les transtypages nécessaires.

Par exemple, la primitive la plus efficace pour instancier dynamiquement une chaîne de caractères est celle qui crée un tableau de caractères, et ceci même en incluant le coût d'un transtypage dans le cas où on veut instancier un type `string` par exemple. C'est d'ailleurs l'implantation utilisée pour le type `string` dans le test de performance de l'instanciation dynamique utilisé à la section 6.1.2.2.

En dehors de `KB`, aucun programme de notre jeu de tests ne fait d'utilisation intensive des chaînes de caractères ; ils représentent une utilisation standard des chaînes, et ont recours aux fonctions de formattage du module `Printf`. Nous avons exclu des tests suivants les programmes ne faisant qu'un usage complètement trivial des chaînes.

	`StringBuilder`	`char[]`	`string`
`Arbl`	1,00	0,97	1,00
`Boyer`	1,00	1,02	1,00
`Derive`	1,25	1,28	1,00
`KB`	1,05	1,05	1,00
`Sieve`	0,99	1,01	1,00

Les résultats sont normalisés sur la représentation `string`. On peut noter que l'écart est faible entre `char[]` et `StringBuilder` et il est difficile de trancher en faveur de l'une ou l'autre des représentations. Remarquons cependant qu'aucun des tests n'effectue de modification en place des chaînes de caractères, domaine où `char[]` a un net avantage d'après la section 6.1.2.2.

En conclusion nous retenons la représentation `char[]` comme étant le choix par défaut pour OCamIL (en raison de ses meilleurs performances pour la mutation de chaînes), même si ce choix n'est pas convaincant sur tous les exemples. Les options du compilateur permettent de changer la représentation des chaînes sans problème, et il sera utile d'utiliser `string` (dans le cas non-mutable) ou `StringBuilder` (dans le cas mutable) pour une meilleure lisibilité des valeurs en phase de débogage.

Évaluation des arguments de fonctions de la droite vers la gauche. Comme indiqué à la section 3.1.3.1, l'ordre d'évaluation des arguments d'une fonction diffère entre les programmes Caml et OCamIL ; il est naturellement de la gauche vers la droite pour ce dernier.

Une option en ligne de commande permet de modifier le comportement par défaut du compilateur OCamIL afin de reproduire le comportement de Caml. Cet aspect de la sémantique de Caml n'étant pas spécifié, nous n'avons pas mis en œuvre une solution très optimisée : celle-ci consiste à insérer dans le langage intermédiaire `Lambda` une série de constructions `let ...in` afin d'extraire et d'évaluer dans l'ordre inverse les arguments des fonctions. Cette implantation introduit un sur-coût qui est mesuré sur le tableau suivant.

`Arbl`	`Avl`	`Bdd`	`Boyer`	`Derive`	`DivEuclid`	`KB`	`Nucleic`	`Quicksort`
1,00	1,75	1,00	1,31	1,00	1,00	1,07	1,06	1,00

`Fft`	`Integrale`	`Pgcd`	`Sieve`	`SoliLet`	`TakE`	`TakU`
1,00	1,00	1,13	1,00	1,00	1,00	12,90

Sans surprise, les performances sont moins bonnes. L'écart le plus grand est enregistré par `TakU`, qui procède exclusivement à des appels de fonctions et des permutations d'arguments.

6.2.3 OCamIL face aux autres compilateurs

6.2.3.1 Performance des objets

Cette section compare la performance des objets CTS et Caml en examinant la rapidité des appels de méthodes d'instance virtuelles[1] et statiques (dans le cas de Objective Caml les méthodes statiques sont de simples fonctions globales). Une même méthode est appelée un milliard de fois, dans des programmes C# et Caml équivalents.

Pour ces différents tests l'environnement d'exécution et la méthode de compilation varient : la colonne C# correspond à une exécution sur la plate-forme .NET (les classes C# sont directement traduites en types CTS), les colonnes `ocamlopt` et `ocamlc` donnent l'exécution sur des objets ayant la représentation propriétaire Caml (respectivement sur la machine à code-octet Caml et en code natif), la colonne `ocamil` correspond à ces mêmes objets mais compilés par OCamIL et tournant sur la plate-forme .NET (n'exploitant pas directement les types CTS, comme expliqué aux sections 3.1.1.3 et 3.1.2.3), et enfin la colonne OJacaré.NET correspond à un interfaçage Caml/C# utilisant l'outil OJacaré.NET (les objets sont en C# mais la boucle d'appel est en Caml). Enfin, il est utile de comparer les performances de OJacaré.NET aux appels équivalents à l'API `Reflection` directement effectués en C# (résultats sur la dernière colonne).

Les appels sont réalisés dans trois cas de figure : au sein d'une même unité de compilation, entre modules Caml distincts ou entre des assemblages différents. Les résultats présentés sur la figure 6.2 sont normalisés sur l'appel de fonction compilé par `ocamlopt`.

Les performances sont disparates. L'appel de méthodes virtuelles sur l'implantation particulière des classes Caml est plus lente à l'origine que le même mécanisme sur la plate-forme .NET (y compris en code natif) et ne passe pas très bien à la compilation OCamIL en termes de performance[2]. De plus dans le cadre d'appels inter-langages, les coûts de sérialisation et le passage par l'API `Reflection` sont d'une magnitude plus élevée que le simple appel de méthode, ce qui explique les résultats de OJacaré.NET (qui n'est toutefois pas loin des coûts cumulés des appels de méthodes Objective Caml et de l'invocation de méthodes par l'API `Reflection`).

Les performances des appels de méthodes statiques sont bien meilleures (affranchies en Caml de la gestion des objets) et comparables entre C#, OCamIL et le code natif engendré par `ocamlopt`.

On remarque aussi qu'il n'y a pas de forte pénalité à effectuer des appels entre

1. C'est-à-dire pouvant être spécialisées dans des classes filles et dont la liaison est tardive.

2. Les performances de OCamIL sont tributaires de celles de Objective Caml puisqu'ils utilisent la même représentation des objets. Cependant en ce qui concerne OCamIL, nous nous trouvons dans un cas d'utilisation de structures génériques dont les performances sont plutôt médiocres. Le prototype OCamIL se base sur la version 3.06 de Objective Caml ; or depuis la représentation des objets a été retravaillée. Nous avons utilisé les tests de cette section pour comparer la performance des appels de méthodes entre les versions 3.06 et 3.09.3 (la dernière en date) du compilateur Objective Caml : on a constaté un gain allant de 0% à 3% maximum.

	C#	`ocamlopt`	`ocamlc`	`ocamil`	OJacaré.NET	`Reflection`
I-mod (I)	2,56	5,25	96,1	339	non(3)	non(4)
X-mod (I)	non(1)	5,25	97,4	339	non(3)	non(1)
X-dll (I)	2,6	non(2)	non(2)	342	1220	574
I-mod (S)	1,04	1,00	79,3	1,17	non(3)	non(4)
X-mod (S)	non(1)	1,00	80	1,46	non(3)	non(1)
X-dll (S)	1,04	non(2)	non(2)	1,49	723	486

Lignes (I) = méthodes d'instance
Lignes (S) = méthodes statiques (fonctions globales dans le cas de Objective Caml)
I-mod = appels internes (intra-module pour Caml et intra-assemblage pour C#)
X-mod = appels inter-modules Caml
X-dll = appels inter-assemblages

Notes :
- (1) N'a de sens qu'en Caml.
- (2) N'a de sens que sur .NET.
- (3) En pur Objective Caml, les performances de OJacaré.NET sont celles de `ocamil`.
- (4) Omis car l'intérêt est ici de comparer `Reflection` à OJacaré.NET.

FIG. 6.2 – *Tests d'appel de méthodes*

modules Caml et entre assemblages .NET.

Bien que les résultats décevants de OCamIL sur les méthodes virtuelles ont un impact en général bien plus faible sur des programmes Caml réels (la majorité du temps est passé dans le code véritable des méthodes et pas dans l'appel lui-même, le style usuel de programmation Caml n'exploite pas les objets intensivement), ils incitent tout de même à concevoir une solution plus adaptée à la plate-forme .NET et au CTS pour la représentation des classes Objective Caml.

Les résultats de OJacaré.NET montrent que ce procédé d'interopération n'est pas très efficace lorsque les composants interfacés communiquent intensivement.

6.2.3.2 Tests comparatifs

Nous comparons dans la suite les performances de OCamIL d'une part avec les compilateurs standard de Caml, à savoir le compilateur de code-octet `ocamlc` et le compilateur de code natif `ocamlopt`, et d'autre part avec d'autres compilateurs visant la plate-forme .NET : F# et SML.NET (bien que ce dernier soit destiné à SML, il nous paraît judicieux de l'inclure dans les tests pour les élargir à un autre membre de la famille ML).

Nous utilisons pour ces tests les dernières versions en date des compilateurs (la version 1.11.7 pour F#, sortie en mai 2006 et la version 1.2 pour SML.NET, sortie en juin 2006). Le compilateur F# est testé avec trois jeux d'options : `fsc` est la version standard, `fsc -Ooff` désactive les optimisations et `fsc -unverifiable` produit une sortie gérée non vérifiable. Les résultats sont normalisés sur `ocamlc`.

	ocamlopt	ocamlc	ocamil	fsc	fsc -Ooff	fsc -unverif	smlnet
Arbl	0,59	1,00	3,79				
Avl	0,16	1,00	0,81				
Bdd	0,06	1,00	0,53	0,41	1,27	0,32	
Boyer	0,21	1,00	11,10	10,41	10,45	9,50	7,67
Derive	0,22	1,00	2,18	1,26	1,30	0,91	
DivEuclid	0,07	1,00	0,68	0,60	0,75	0,24	
Fft	0,12	1,00	0,47	0,52	0,86	0,52	
Integrale	0,48	1,00	2,04	2,41	3,75	2,41	
KB	0,15	1,00	17,85	13,85	14,56	10,95	9,52
Morematch	0,048	1,00	0,746	0,523	0,634	0,35	
Morematch2	0,044	1,00	1,006				
Nucleic	0,19	1,00	0,79	0,49	0,79	0,47	
Pgcd	0,33	1,00	0,65	0,24	1,46	0,24	
Quicksort	0,03	1,00	0,21	0,43	2,44	0,42	
Sieve	2,40	1,00	5,21				
SoliLet	0,12	1,00	0,90	0,43	0,48	0,24	
Syrac	0,08	1,00	0,61	0,10	2,37	0,10	
Tailcall	0,08	1,00	0,54	0,03	1,00	0,03	
TakE	0,23	1,00	37,20	30,36	31,60	30,48	31,37
TakC	0,05	1,00	0,27	0,09	1,61	0,09	0,10
TakU	0,03	1,00	0,17	0,06	1,01	0,06	0,06

Il a fallu traduire les programmes de test vers Standard ML, ce qui n'a été fait que pour quelques-uns. D'autre part, le compilateur F# n'accepte pas tous les programmes Caml, ce qui explique l'absence de test pour `Arbl` (qui utilise des modules et foncteurs) et pour `Morematch2` (les variants polymorphes n'existent pas en F#). `Sieve` manque car sa version F# produit un dépassement de pile, et `Avl` ne compile pas correctement (bogue dans le typeur de F#).

Les tests montrent que les trois compilateurs .NET n'ont pas de bons résultats sur les programmes très fonctionnels (`KB` et `Boyer` : ce sont les écarts les plus importants avec `ocamlopt`, les facteurs étant proches de 100 et 50 respectivement) ou utilisant les exceptions comme structure de contrôle (`TakE`). On peut remarquer que pour ces exemples de performances très médiocres, tous les compilateurs .NET enregistrent des résultats similaires.

En revanche, les calculs entiers et flottants monomorphes donnent des résultats bien meilleurs. Les performances de OCamIL sur le filtrage de motifs sont bonnes (test `Morematch`), y compris sur les variants polymorphes (test `Morematch2`). Plusieurs exemples donnent des résultats meilleurs sur la plate-forme .NET qu'avec la machine à code-octet Caml : `Avl`, `Bdd`, `DivEuclid`, `Fft`, `Morematch`, `Nucleic`, `Quicksort`, etc. . . , parmi lesquels se trouvent presque toutes les « vraies » applications, ce qui est encourageant. En raison de ses optimisations et de sa monomorphisation globale, le compilateur SML.NET arrive en tête des performances sur les tests auxquels il participe.

On peut déduire de la comparaison avec F# et SML.NET que les performances de OCamIL peuvent encore être améliorées. La bonne qualité du front-end (comme par exemple la compilation élaborée du filtrage de motifs) est pénalisée par les structures de données utilisées par le back-end. Certaines analyses statiques permettraient de faire gagner davantage en performance, par l'optimisation des représentations des chaînes de caractères et des exceptions dans des zones localisées du code produit.

La production de code géré non-vérifiable n'a pas été expérimentée avec OCamIL mais devrait produire de bons résultats. La suite du chapitre présente les améliorations qui ont été introduites pour le code géré vérifiable.

6.2.3.3 Amélioration des performances

Il nous faut comprendre de manière approfondie les écarts de performances entre OCamIL et F# quand ils sont nettement en faveur de ce dernier. Le désassembleur de code CIL `ildasm` permet de comprendre les optimisations utilisées.

Les trois versions du programme `Tak` sont très simples et donnent pourtant des résultats différents. L'examen de `TakU/C` montre que le gain provient de l'implantation de la récursion terminale : alors que OCamIL utilise toujours l'instruction `.tail` et un appel de fonction, les autres compilateurs optimisent lorsque c'est la même fonction qui s'appelle elle-même lors de la récursion terminale : dans ces cas ils utilisent des instructions `starg` pour écraser les arguments et branchent sur le début du code de la méthode au lieu d'effectuer un appel. Cette optimisation peut facilement être implantée dans OCamIL.

En ce qui concerne `TakE`, le gain provient à la fois de la récursivité terminale évoquée plus haut et de la représentation des exceptions : OCamIL utilise des blocs à la manière de Caml, où une chaîne de caractères identifie l'exception, alors que ses concurrents définissent des classes dédiées. La construction d'une exception est alors plus simple ainsi que le filtrage (reposant sur l'instruction `isinst`), rendant les blocs de traitement des exceptions plus légers.

Nous avons implanté l'optimisation des appels récursifs terminaux dans OCamIL et refait les tests. Les améliorations sont sensibles sur `Bdd`, `Pgcd` (gain $\times 2$), `Syrac` (gain $\times 3{,}6$), `Tailcall` (gain $\times 11$), `TakC` et `TakU` (gain $\times 2$). Bien sûr ces tests ne ressemblent en rien à des programmes réels pour lesquels les bénéfices de l'optimisation sont dilués.

	`ocamlopt`	`ocamlc`	`ocamil`	`fsc`	`fsc -Ooff`	`fsc -unverif`	`smlnet`
`Arbl`	0,59	1,00	3,79				
`Avl`	0,16	1,00	0,70				
`Bdd`	0,06	1,00	0,48	0,41	1,27	0,32	
`Boyer`	0,21	1,00	11,10	10,41	10,45	9,50	7,67
`Derive`	0,22	1,00	2,18	1,26	1,30	0,91	
`DivEuclid`	0,07	1,00	0,68	0,60	0,75	0,24	
`Fft`	0,12	1,00	0,46	0,52	0,86	0,52	
`Integrale`	0,48	1,00	2,04	2,41	3,75	2,41	
`KB`	0,15	1,00	17,83	13,85	14,56	10,95	9,52
`Morematch`	0,048	1,00	0,746	0,523	0,634	0,35	
`Morematch2`	0,044	1,00	1,006				
`Nucleic`	0,19	1,00	0,76	0,49	0,79	0,47	
`Pgcd`	0,33	1,00	0,34	0,24	1,46	0,24	
`Quicksort`	0,03	1,00	0,20	0,43	2,44	0,42	
`Sieve`	2,40	1,00	5,2				
`SoliLet`	0,12	1,00	0,88	0,43	0,48	0,24	
`Syrac`	0,08	1,00	0,17	0,10	2,37	0,10	
`Tailcall`	0,08	1,00	0,05	0,03	1,00	0,03	
`TakE`	0,23	1,00	37,14	30,36	31,60	30,48	31,37
`TakC`	0,05	1,00	0,12	0,09	1,61	0,09	0,10
`TakU`	0,03	1,00	0,08	0,06	1,01	0,06	0,06

Les performances sont proches de celles de F#, tantôt supérieures, tantôt inférieures. Des progrès sont encore possibles pour `KB` (pour ce dernier on a vu à la section 6.2.2.2 que l'utilisation de chaînes de caractères non mutables permet de gagner 5%, ce qui le ramène aux performances de la version F#), `TakE` (en optimisant la représentation des exceptions), `Derive`, `Morematch`, `Nucleic` et `SoliLet`.

Pour ces derniers une analyse plus poussée à l'aide d'un profileur révèle que l'occupation mémoire et l'action du ramasse-miettes entravent les performances. La version F# utilise des classes plus légères pour l'implantation des variants (les instances ne comportent d'entier de tag, au lieu de cela une fonction `getTag` le calcule à partir d'instructions `isinst` destinées à retrouver le type exact de l'instance considérée). Dans le cas d'un test de performances qui pousse à la création de nombreuses instances de classes, le gain de place devient sensible et limite les activations du ramasse-miettes. D'autre part F# utilise des classes dédiées pour les listes (très nombreuses dans ces tests) ainsi que pour les n-uplets (il existe des classes spécifiques dans la bibliothèque de support de F# pour les n-uplets ayant un faible nombre de composants). Il est tout à fait envisageable d'implanter des optimisations similaires dans OCamIL. Même dans le cas de n-uplets polymorphes implantés par des classes dont les champs sont typés par `object`, on obtient des structures plus efficaces que des tableaux d'objets nécessitant des tests de bornes.

6.2.3.4 Temps de compilation

Le test suivant donne le temps de compilation d'un programme Caml d'environ 3000 lignes. Cela permet de comparer OCamIL aux compilateurs INRIA (tous s'exécutant sous la forme de code-octet Caml) ainsi qu'avec le compilateur OCamIL sous forme bootstrappée.

`ocamlc`	`ocamlopt`	`pre-ocamil`	`ocamil`
0,448	1,00	1,07	7,33

On voit que la charge des passes de compilation de OCamIL est comparable à celle du compilateur INRIA de code natif. En revanche la performance du code OCamIL face au code-octet Caml lorsque l'exécutable est le compilateur Caml lui-même (colonnes `pre-ocamil` et `ocamil`, voir aussi le dispositif de bootstrap à la section 5.2.1.1) est moyenne. Les sources d'inefficacité isolées dans les sections précédentes (style très fonctionnel, structures de données polymorphes et utilisation des exceptions) s'appliquent au code du compilateur lui-même.

En conclusion de ce chapitre, on peut dire que si certaines optimisations permettraient d'améliorer les performances du compilateur OCamIL, celui-ci occupe déjà une place raisonnable dans la gamme des compilateurs de langages fonctionnels sur la plate-forme .NET. Il apparaît clairement que sur les programmes très fonctionnels, utilisant des structures de données polymorphes et les exceptions comme mécanisme de contrôle, les performances ne sont jamais bonnes, et proviennent de lacunes de la plate-forme subsistant dans ces domaines. Afin de mieux traiter les problèmes de performance, il faudrait mener des tests permettant de séparer le temps d'exécution du temps de récupération de la mémoire (un point où Objective Caml est reconnu comme excellent) : cela permettra de décomposer les efforts d'optimisation en deux catégories distinctes (optimisation du contrôle et de la taille des données).

Certaines des techniques utilisées par d'autres compilateurs .NET devraient permettre d'améliorer les performances de OCamIL. Des analyses statiques comme décrites dans [62] peuvent être intéressantes pour contrôler la représentation des exceptions. D'autre part si on peut s'inspirer de SML.NET pour mettre en œuvre une politique de monomorphisation des valeurs, on favorisera plutôt dans le cas de .NET l'utilisation des Generics, qui permettent une monomorphisation dynamique (au moment de la compilation *Just In Tine*) directement prise en charge par l'environnement d'exécution. La production de code géré non-vérifiable (comme le propose F#) laisse espérer quant à elle une amélioration des performances d'environ 15%.

Conclusion

Nous voyons le travail de cette thèse comme une contribution au rapprochement de deux communautés qui ont tout intérêt à ne pas s'ignorer.

La communauté des langages fonctionnels est difficilement dissociable des milieux académiques. Ces langages bénéficient d'une assise théorique solide qui continue toujours à s'enrichir, et ont créé un terrain propice à l'innovation et à l'expérimentation. Bien présents dans l'enseignement universitaire, ils forment pourtant une proportion minoritaire des langages utilisés dans l'industrie. De nombreux apports des langages fonctionnels (en matière de typage, mais aussi de modularité) ne sont reconnus que tardivement et incorporés, timidement, dans les langages formant l'avant-scène de l'industrie logicielle.

D'un autre côté, la voie ouverte par Java et suivie par C# suscite beaucoup d'intérêt de la part des programmeurs. Ces langages et leurs plates-formes d'exécution respectives donnent l'accès à des services de haut niveau d'une manière simple et tirent profit de mécanismes de gestion de code, de mémoire et de sécurité facilitant le travail du programmeur. Ils ont de plus rapidement conquis le secteur incontournable des langages de script du Web (Java EE et ASP.NET). Ces langages ont une architecture reposant sur des machines virtuelles désormais largement diffusées, suivant ainsi la même approche que plusieurs implantations de langages fonctionnels. Compiler ces derniers sur ces machines virtuelles répandues permet d'accélérer leur diffusion et ouvre la porte à une interopérabilité aisée.

Cette thèse s'intéresse à ces questions sous un angle pratique, aussi avons-nous fait le choix d'un langage fonctionnel (Objective Caml) et d'une plate-forme (.NET) pour mener à bien nos expérimentations. Les apports de ce travail sont les suivants :

- La réalisation d'un compilateur complet (supportant la totalité des traits du langage source, y compris la couche objet et le système de modules et foncteurs) et performant (offrant une qualité semblable aux projets industriels tels que F#). Le compilateur a été boostrappé et tourne donc comme un composant .NET indépendant. Le toplevel Caml a été adapté avec succès : il tourne sous .NET et utilise la bibliothèque `Reflection` pour générer dynamiquement le code compilé en mémoire.
- L'investigation de deux solutions antagonistes au problème des informations de types devant combler le vide laissé entre d'une part un typeur ne propageant pas ces informations (sur la base d'optimisations permises par un langage

statiquement typé comme Caml) et d'autre part un environnement d'exécution typé. La première solution a recherché la modularité à tout prix et a montré qu'il est possible de reconstruire une information suffisante pour générer des programmes corrects sur la plate-forme .NET, dans le cadre d'un projet à court terme (de l'ordre d'un an de développement). La deuxième solution s'est employée à propager les informations de types manquantes afin d'obtenir des programmes plus efficaces.

– La conception et réalisation de l'outil OJacaré.NET rend possible l'interopération entre des composants écrits en Objective Caml et d'autres compilés à partir de langages tiers disponibles pour la plate-forme .NET. Ce travail montre une manière de concilier deux modèles de classes différents, les utiliser pour communiquer, et tirer avantage des deux. Les capacités de OJacaré.NET sont démontrées par son emploi au sein même du compilateur OCamIL (binding avec l'API `Reflection`). De plus le couple OCamIL/OJacaré.NET résout des problèmes rencontrés par le prédécesseur de ce dernier, OJacaré, qui était contraint de faire cohabiter les environnements d'exécution Java et Caml.
– Une sérialisation sûre s'appuyant les informations de typage CTS. La désérialisation n'échoue jamais et l'utilisation d'une valeur dé-sérialisée sous un type autre que le sien va provoquer le lancement d'une exception, qu'il est possible de rattraper dans le programme Caml.
– Ce travail permet enfin d'évaluer la pertinence d'une plate-forme comme .NET comme environnement d'exécution pour d'autres langages, en terme de richesse et de performances.

Revenons sur les deux approches de reconstruction et propagation des types. La reconstruction est moins intrusive et reproduit plus fidèlement l'environnement d'exécution standard de Caml, qui est non-typé (pour le code-octet Caml comme pour le code natif). L'absence de contraintes de typage permet en outre d'implanter facilement les constructions Caml les plus complexes (par exemple il n'y a pas de limitation aux redéfinitions de types dans des déclarations fonctorielles comme expliqué dans la section 3.1.4).

D'un autre côté la propagation des types nécessite de modifier le compilateur Objective Caml de l'INRIA plus haut dans la chaîne de compilation, celui-ci éliminant le plus tôt possible les informations de types. À nos yeux c'est là un défaut du compilateur Caml en matière d'ouverture sur le monde extérieur et le projet OCamIL introduit un début de solution. La propagation des types permet les choix de représentations des valeurs les plus fins, ce qui est garant de bonnes performances, mais aussi de lisibilité et de pertinence du code engendré, facilitant ainsi l'implantation d'un débogueur (avoir des types précis à l'exécution permet une exploration des valeurs pertinente en session de débogage) et améliorant les perspectives d'interopération.

La plate-forme .NET n'est pas aujourd'hui totalement adaptée aux langages fonctionnels. Un premier pas a été franchi en permettant la récursion terminale, mais les fermetures ne trouvent pas encore de représentation satisfaisante, comme le prouvent nos tests de performances. Cela dit les extensions Generics (faisant dorénavant partie de la plate-forme standard) et ILX vont dans la bonne direction. D'autre part, on ne peut pas dire non plus que les langages fonctionnels, en tout cas à travers l'exemple de Objective Caml, puissent facilement s'ouvrir sur l'extérieur. En particulier aller au bout du credo du typage statique jusqu'à perdre les informations de types dans un souci d'optimalité contribue à isoler ces langages.

Le bilan de l'expérience reste tout de même très positif, en particulier par les possibilités offertes par l'interopération mais aussi par un certain nombre d'avantages immédiats découlant de l'adaptation de Objective Caml à la plate-forme .NET : l'utilisation de threads et d'un GC concurrent mis à disposition par plate-forme d'exécution (avec la possibilité de tirer partie d'architectures multi-processeurs), des avantages liés à l'introspection de types sous .NET et une sérialisation sûre. Ces différents résultats s'appuient sur la génération de code géré ; générer du code natif ne l'aurait pas permis et aurait demandé de manipuler deux environnement d'exécution différents, compliquant d'autant l'interopérabilité. Une voie médiane consiste à générer du code géré non-vérifiable en profitant des garanties du typage statique de Caml. À partir des résultats de F#, on peut alors compter sur une amélioration des performances de 10%-15% (sur la base de notre jeu de tests).

Les directions suivantes permettent de prolonger le travail de cette thèse :

- Implanter les optimisations suggérées par les tests de performances, principalement des analyses statiques portant sur les exceptions (les exceptions natives de la plate-forme .NET sont très commodes, et doivent être conservées, dans le cadre de l'interopération, mais sont beaucoup trop coûteuses lorsqu'elles sont utilisées comme structure de contrôle dans des algorithmes très fonctionnels).
- Exploiter le polymorphisme offert par les Generics. Cette notion de polymorphisme permet d'implanter plus efficacement des algorithmes génériques sans mettre en œuvre une politique de monomorphisation dans le compilateur. La monormorphisation est assurée dynamiquement par la plate-forme cible et permet de s'affranchir du recours permanent au type `object` dans la déclaration de structures de données polymorphes. Cela va éliminer un grand nombre d'opérations d'encapsulation et améliorer les performances.
- Enrichir l'IDL de OJacaré.NET avec de nouvelles constructions : les types délégués et les expressions de types des Generics. En plus des avantages mentionnés ci-dessus, les Generics fournissent un moyen naturel d'implanter des données et des algorithmes polymorphes sur .NET, dans un cadre convenable pour différents langages. Permettre l'interopérabilité entre langages fonctionnels et des langages comme C# s'appuyant sur les Generics est un sujet de recherche prometteur.
- Proposer un mode permettant de générer du code natif, et en général renforcer la communication avec le monde non géré (composants externes en code natif ou interfacés par COM). Le but principal est d'augmenter encore davantage l'ouverture de Caml, en lui donnant accès au monde natif par le biais de la plate-forme .NET. Il est également intéressant d'investiguer des représentations mixtes gérées/non-gérées pour la compilation de Objective Caml (à la manière

de Visual C++ avec extensions gérées) pour gagner en rapidité d'exécution. Cependant il est assez complexe de gérer cette mixité dans un cadre sûr, en présence d'un récupérateur automatique de mémoire, et en même temps de conserver de bonnes propriétés d'interopérabilité. Enfin il serait intéressant de voir comment l'infrastructure .NET peut apporter à OCamIL le chargement dynamique de code natif.
- Développer un back-end pour Java. Le travail réalisé sur .NET peut être facilement adapté à la JVM. Il est envisagé de paramétrer les transformations effectuées par le compilateur OCamIL de manière à pouvoir cibler l'une ou l'autre des machines virtuelles.
- Intégrer OCamIL dans les outils de débogage de la plate-forme .NET. Le code cible peut être enrichi d'attributs contenant des informations de débogage. Le travail supplémentaire est alors de propager des informations de code source jusqu'au code compilé, ce qui permet de coupler l'arbre de syntaxe typé avec le code exécuté, offrant ainsi l'exploration des variables locales typées, la gestion de points d'arrêts liés au code source etc... Les attributs de débogage étant standard, on peut alors utiliser directement les débogueurs .NET sur des programmes OCamIL. La propagation des types et leur présence à l'exécution .NET permet automatiquement une exploration intuitive des valeurs typées (une nouveauté pour les environnements de développement Caml).
- Intégrer OCamIL dans des IDE comme Eclipse et Visual Studio. L'idée est d'interfacer les analyseurs lexicaux et syntaxiques ainsi que le typeur de Caml dans ces environnement de développements, pour obtenir une aide contextuelle riche et une inférence de types en temps réel, et un cadre confortable pour le débogage. Cela a été réalisé avec succès par SML.NET, et est en cours de développement pour F#.
- Pour améliorer la sérialisation sûre, il est possible de confronter les valeurs dé-sérialisées à une expression de types à la manière de [47], afin d'obtenir une détection anticipée des échecs de transtypage. L'implantation peut s'appuyer sur l'API `Reflection`.

Nous espérons que ce travail pourra fournir d'utiles résultats et de futures pistes dans le domaine de la compilation de langages fonctionnels statiquement typés vers une machine virtuelle typée : notamment la préservation des informations de typage, qui pour la catégorie de plates-formes visées est garante de performances, mais aussi d'ouverture sur le monde extérieur en engageant le langage source sur la voie de l'interopérabilité. On peut dire de la plate-forme .NET qu'elle n'est pas encore tout à fait adaptée aux constructions trouvées dans ces langages : une notion de fermeture native est toujours manquante, toutefois les progrès enregistrés dans le domaine du polymorphisme (tels que les Generics de .NET 2.0) incitent à l'optimisme. La machine virtuelle idéale devra être multi-langages et portable, combinant les avantages de .NET et de la JVM, avec une meilleure prise en charge des langages fonctionnels.

Appendices

Le code-octet CIL

Les notations utilisées ont recours aux conventions suivantes :
- les crochets [] désignent un suffixe optionnel (ainsi `div[.un]` fait référence à deux instructions : `div` et `div.un`),
- les accolades {} désignent une énumération (ainsi `conv{i,u}` fait référence aux deux instructions `convi` et `convu`).

Il est possible de les combiner : par exemple `add[{.ovf,.ovf.un}]` désigne les trois instructions `add`, `add.ovf` et `add.ovf.un`.

Nous donnons ici le jeu d'instructions de la machine .NET dans sa version 1.1. La version 2.0 introduit des opérations supplémentaires pour gérer les Generics (la plate-forme 3.0, quant à elle, se base sur la machine précédente 2.0).

Opérations numériques

Calculs	Conversions
`add[{.ovf,.ovf.un}]`	`conv{i,u}{1,2,4,8}`
`sub[{.ovf,.ovf.un}]`	`conv{i,u}`
`mul[{.ovf,.ovf.un}]`	`conv.ovf.{i,u}{1,2,4,8}[un]`
`div[.un]`	`conv.ovf.{i,u}[un]`
`rem[.un]`	`conv.r{4,8}`
`neg`	`conv.r.un`
`ckfinite`	
`and`, `or`, `xor`, `not`, `shl`, `shr[.un]`	
`ceq`, `cgt[.un]`, `clt[.un]`	

Les opérations arithmétiques existent sous plusieurs versions, selon que l'on veut détecter les dépassements de capacité (suffixe `ovf`) et ne pas tenir compte du signe (suffixe `un`). Les opérations ne sont pas typées et on utilise les mêmes instructions pour agir sur les différents types d'entiers et de flottants. Les mêmes remarques s'appliquent aux comparaisons et aux conversions.

Les conversions visent les différents types entiers : occupant 1, 2, 4, 8 octets, sous forme signée ou non, ainsi que les entiers natifs de la plate-forme hôte sous forme signée ou non (`conv.i` et `conv.u`).

La machine dispose également d'opérations booléennes. Pour terminer l'instruction `ckfinite` teste si une valeur flottante est définie et finie (c'est-à-dire différente de `NaN` et ±`Infty`).

Opérations de pile

Vers la pile	Depuis la pile
Constantes :	

`ldc.i{4,8,4.s}`
`ldc.i4.{0,1,2,3,4,5,6,7,8,m1}`
`ldc.r{4,8}`	

Arguments :	

`ldarg[.s]`
`ldarg.{0,1,3,4}`
`ldarga[.s]`
	`starg[.s]`

Variables locales :	

`ldloc[.s]`
`ldloc.{0,1,2,3}`
`ldloca[.s]`
	`stloc[.s]`
	`stloc.{0,1,2,3}`

Par indirections :	

`ldind.{i,u}{1,2,4,8}`,
`ldind.i`
`ldind.r{4,8}`
`ldind.ref`
	`stind.i{1,2,4,8}`, `stind.i`
	`stind.r{4,8}`
	`stind.ref`

Pointeurs de méthodes :	
`ldftn`, `ldvirtftn`	
Objets :	
`ldobj` `ldnull`, `ldstr`	`stobj`
Champs d'un objet :	
`ldfld`, `ldsfld` `ldflda`, `ldsflda`	`stfld`, `stsfld`
Éléments d'un tableau :	

`ldelem.{i,u}{1,2,4,8}`,
`ldelem.i`
`ldelem.r{4,8}`
`ldelem.ref`
`ldelema`
	`stelem.i{1,2,4,8}`, `stelem.i`
	`stelem.r{4,8}`
	`stelem.ref`

Autres :	

`arglist`
`ldlen`
`ldtoken`
`dup`
	`pop`

La plupart des instructions d'empilement ou de dépilement des valeurs sont annotées par des types et admettent des cas particuliers optimisés :

- On distingue les types entiers de 4 et 8 octets signés ou non, les entiers natifs avec ou sans signe, ainsi que les flottants en simple et double précision.
- Les optimisations rendent les instructions plus compactes : par exemple alors que `ldc.i4` doivent être complétées par l'entier 32 bits à empiler, `ldc.i4.0` est

spécifique à l'entier 0. L'instruction `ldc.i4.s` prend un entier en argument, mais limité à l'intervalle $-128 \ldots 127$, et n'occupant donc qu'un seul octet au lieu de 4 dans le flot d'instructions.

Les instructions `ldc` empilent des constantes. Les instructions `ldloc`/`stloc` empilent/dépilent des valeurs depuis les variables locales. Les instructions `ldarg`/`starg` font de même avec les arguments de la méthode. Dans les deux cas l'index de la variable est soit intégrée à l'instruction sous forme d'une constante, soit fait partie d'une instruction optimisée (comme `ldloc.0` qui empile la première variable locale, ou `ldloc.s` dont l'index n'occupe qu'un octet).

Les instructions `ldfld` et `ldsfld` empilent des champs d'instance et de classe. Les instructions `ldelem` empilent des champs de tableau : ces dernières intègrent le type de la donnée à empiler (dont `.ref`, utilisé pour les types référence). Les instructions `ldind` dé-référencent des pointeurs. L'instruction `ldobj` empile un type valeur en suivant un pointeur. Chacune des familles d'instructions précédentes possède aussi des instructions de dépilement.

L'instruction `ldnull` empile la valeur `null` et `ldstr` empile une chaîne de caractères constante. L'instruction `ldlen` calcule la longueur d'une chaîne de caractères, `dup` duplique la valeur en sommet de pile alors que `pop` l'élimine.

Les instructions de la forme `ld*a` empilent l'adresse d'une valeur au lieu de la valeur elle-même.

Restent deux instructions spéciales : `arglist` empile un pointeur vers les arguments d'une méthode ayant un nombre variable d'arguments et `ldtoken T` empile un jeton de méta-données `T` (cette instruction est utile pour interagir avec l'API `Reflection`).

Contrôle

Branchement	Appels	Exceptions

`br[.s]`
`brfalse[.s]`,
`brtrue[.s]`
`beq[.s]`
`bne.un[.s]`
`bge[.un][.s]`,
`bgt[.un] [.s]`,
`ble[.un] [.s]`,
`blt[.un] [.s]`
`switch`
	`jmp`
	`call`
	`calli`
	`callvirt`
	`ret`
		`throw`
		`rethrow`
		`leave[.s]`
		`endfilter`
		`endfinally`

Parmi les instructions de contrôle se comptent les branchements conditionnels et inconditionnels, y compris l'instruction `switch`. Les étiquettes de saut sont encodées directement dans les instructions sous forme d'entiers signés (des sauts relatifs) occupant 4 octets, ou 1 seul octet dans les formes optimisées `.s`.

Les instructions d'appels `call` et `callvirt` diffèrent : la première effectue un appel de méthode non virtuel qui court-circuite la liaison tardive. L'instruction `calli` effectue un appel indirect à travers un pointeur de méthode. L'instruction `jmp` fait passer l'exécution à une autre méthode, en prenant pour arguments ceux de la méthode courante.

Les autres instructions sont consacrées aux exceptions. Les blocs d'exceptions ne sont pas définis par des instruction mais par des méta-données particulières qui marquent des segments dans le flot d'instructions de la méthode.

Mémoire

Allocation	Initialisation	Copie	Taille

`localloc`
`newarr`
`newobj`
	`initblk`
	`initobj`
		`cpblk`
		`cpobj`
			`sizeof`

Nous regroupons ici des fonctions d'allocation : `newobj` et `newarr` construisent des objets et des tableaux, `initobj` et `cpobj` initialisent et copient des types valeurs.

Les instructions `initblk` et `cpblk` allouent et copient des plages de données ; `localloc` alloue une plage de mémoire dans la zone dynamique de mémoire locale de la méthode. Ces trois instructions ne sont pas autorisées dans du code vérifiable.

Enfin `sizeof` donne la taille en octets d'un type valeur.

Autres opérations (objet)

Objets / Value types	Références typées

`castclass`
`isinst`
`box`
`unbox`
	`mkrefany`
	`refanyval`
	`refanytype`

L'instruction `castclass` tente de modifier le type dynamique d'un objet et `isinst` teste si un objet est d'un type donné.

L'instruction `box` et `unbox` servent à l'encapsulation et la désencapsulation des types valeurs.

L'instruction `mkrefany` fabrique une référence typée alors que `refanyval` et `refanytype` permettent d'extraire d'une référence typée sa valeur et son type. Les références typées ne sont utilisées que par Visual Basic.Net à notre connaissance.

Divers

Instructions	Préfixes

`nop`
`break`
	`unaligned.`
	`volatile.`
	`tail.`

L'instruction `nop` n'effectue rien et `break` sert à marquer le code à destination des débogueurs.

Pour terminer, des instructions spéciales servent à préfixer d'autres instructions :

- `unaligned`. indique qu'une adresse n'est pas forcément alignée en mémoire,
- `volatile.` indique qu'une variable est volatile,
- `tail.` indique qu'un appel est effectué de manière récursive terminale.

Les primitives de Objective Caml

Primitives des langages `Lambda` et `Clambda`

Nous détaillons ici les primitives utilisées dans le compilateur Caml au niveau des langages intermédiaires `Lambda` et `Clambda`. Rappelons que les paramètres font partie de la primitive (ils sont statiques) et que les arguments peuvent être dynamiques, résultant de l'évaluation d'expressions Caml.

Opérations sur les entiers, les booléens et les flottants			
Primitive	**Paramètres**	**Arguments**	**Description**
`Paddint`		`t1` et `t2` entiers	addition entière `t1` + `t2`
`Psubint`		`t1` et `t2` entiers	soustraction entière `t1` −`t2`
`Pnegint`		`t` entier	opposé entier − `t`
`Pmulint`		`t1` et `t2` entiers	multiplication entière `t1` × `t2`
`Pdivint`		`t1` et `t2` entiers	division entière `t1` / `t2`
`Pmodint`		`t1` et `t2` entiers	modulo entier `t1` % `t2`
`Pandint`		`t1` et `t2` entiers	ET binaire `t1` & `t2`
`Porint`		`t1` et `t2` entiers	OU binaire `t1` | `t2`
`Pxorint`		`t1` et `t2` entiers	OU exclusif binaire `t1` ^ `t2`
`Plslint`		`t1` et `t2` entiers	décalage logique à gauche `t1` << `t2`
`Plsrint`		`t1` et `t2` entiers	décalage logique à droite `t1` >>> `t2`
`Pasrint`		`t1` et `t2` entiers	décalage arithmétique à droite `t1` >> `t2`
`Pintcomp`	comparaison C	`t1` et `t2` entiers	comparaison entière `t1` C `t2`
`Poffsetint`	un entier i	`t` un terme entier	calcul de `t`+i
`Poffsetref`	un entier i	`t` correspondant à une référence sur un entier	opération `t:=!t`+i
`Psequand`		`t1` et `t2` booléens	ET logique (séquentiel) `t1` && `t2`
`Psequor`		`t1` et `t2` booléens	OU logique (séquentiel) `t1` || `t2`
`Pnot`		`t` booléen	NON logique ∼`t`
`Paddfloat`		`t1` et `t2` flottants	addition flottante `t1` + `t2`
`Psubfloat`		`t1` et `t2` flottants	soustraction flottante `t1` − `t2`
`Pnegfloat`		`t` flottant	opposé flottant − `t`
`Pabsfloat`		`t` flottant	valeur absolue flottante |`t`|
`Pmulfloat`		`t1` et `t2` flottants	multiplication flottante `t1` × `t2`
`Pdivfloat`		`t1` et `t2` flottants	division flottante `t1` / `t2`
`Pfloatcomp`	comparaison C	`t1` et `t2` flottants	comparaison flottante `t1` C `t2`
`Pintoffloat`		`t` flottant	conversion flottant vers entier
`Pfloatofint`		`t` entier	conversion entier vers flottant

Notes :

- La comparaison C peut être : $=$, $\neq$, $<$, $\leq$, $>$ ou $\geq$.
- Les booléens ne forment pas un type séparé des entiers, mais les opérations logiques entières et booléennes sont nécessairement différenciées.

Manipulation de blocs			
Primitive	**Paramètres**	**Arguments**	**Description**
`Pmakeblock`	entier i et flag de mutabilité	`t_1` ... `t_n` n termes quelconques	création d'un bloc de tag i constitué des n valeurs
`Pfield`	entier i	`t` un bloc	lecture `t`[i]
`Psetfield`	entier i et booléen (cf notes)	`t1` un bloc et `t2` un terme	écriture `t1`[i]←`t2`
`Pfloatfield`	entier i	`t` un bloc	lecture `t`[i]
`Psetfloatfield`	entier i	`t1` un bloc et `t2` un terme flottant	écriture `t1`[i]←`t2`

Notes :

- Le booléen indiquant la mutabilité dans `Pmakeblock` n'entre pas en jeu dans la compilation de cette primitive.
- La lecture et l'écriture dans un bloc sont nécessairement sûres (car l'index en paramètre est déterminé statiquement : voir la différence avec les chaînes de caractères ou les tableaux plus bas).
- Le booléen paramètre de `Psetfield` est vrai lorsque le terme `t2` est susceptible d'être un pointeur sur un bloc, auquel cas la primitive doit se soucier d'éventuellement signaler au ramasse-miettes la nouvelle capture du pointeur.
- Les enregistrements ne contenant que des flottants sont représentés à la manière des tableaux de flottants et ces blocs possèdent des primitives d'accès dédiées. Ces dernières sont utilisées quant le typage a déterminé statiquement que l'argument aura cette représentation spécifique. Les fonctions polymorphes font appel aux primitives génériques, qui doivent savoir traiter toutes les représentations (cela suppose un examen des blocs à l'exécution).

Manipulation de chaînes de caractères		
Primitive	**Arguments**	**Description**
`Pstringrefu`	`t1` chaîne et `t2` entier	lecture *unsafe* `t1`[`t2`]
`Pstringrefs`	`t1` chaîne et `t2` entier	lecture *safe* `t1`[`t2`]
`Pstringsetu`	`t1` chaîne, `t2` entier et `t3` caractère	écriture *unsafe* `t1`[`t2`]←`t3`
`Pstringsets`	`t1` chaîne, `t2` entier et `t3` caractère	écriture *safe* `t1`[`t2`]←`t3`
`Pstringlength`	`t` chaîne	longueur de la chaîne `t`

Notes :

- Les primitives d'accès aux chaînes de caractères existent sous deux versions : une version sûre qui lève une exception en cas d'indice incorrect et une version non sûre plus rapide, mais pour laquelle il n'y a pas de mécanisme de rattrapage d'erreur (le programme s'arrête sur une erreur irrécupérable).
- Les caractères Caml sont des caractères 8 bits.

– Les chaînes de caractères de Caml ne se terminent pas obligatoirement par un caractère nul, puisque la taille est contenue dans l'en-tête du bloc de chaîne.

Manipulation de tableaux			
Primitive	**Paramètres**	**Arguments**	**Description**
`Pmakearray`	type de tableau	des termes `t_1`...`t_n`	
`Parraylength`	type de tableau	`t` un tableau	longueur du tableau
`Parrayrefu`	type de tableau	`t1` tableau et `t2` entier	lecture *unsafe* `t1[t2]`
`Parrayrefs`	type de tableau	`t1` tableau et `t2` entier	lecture *safe* `t1[t2]`
`Parraysetu`	type de tableau	`t1` tableau, `t2` entier et `t3` terme quelconque	`t1[t2]`←`t3`
`Parraysets`	type de tableau	`t1` tableau, `t2` entier et `t3` terme quelconque	écriture *safe* `t1[t2]`←`t3`

Notes :

– Tout comme pour les chaînes de caractères, les primitives d'accès aux tableaux existent sous versions : sûre et non sûre.

– Le type de tableau permet de savoir si les éléments sont susceptibles d'être surveillés par le ramasse-miettes : les tableaux constitués seulement d'entiers ou de flottants ne le nécessitent pas par exemple.

Primitives liées au contrôle			
Primitive	**Paramètres**	**Arguments**	**Description**
`Praise`		`t` un terme représentant une exception	déclenche l'exception
`Pccall`	description de la primitive externe	des termes `t_1`...`t_n`	appel de la primitive avec les arguments `t_1`...`t_n`
`Pisint`		`t` un terme quelconque	indique si le terme est un entier ou un bloc
`Pisout`		`t1` et `t2` deux termes entiers	détermine si `t1`<`t2`

Notes :

– La description de la primitive externe permet d'effectuer un appel vers une primitive C au sein de l'environnement d'exécution ou bien fournie par l'utilisateur au moment de l'édition de lien.

– `Pisint` est utilisée pour le filtrage de motif, dans l'analyse d'une valeur de type variant.

– `Pisout` est une version optimisée de l'inégalité entière utilisée pour la compilation du filtrage de motif.

Entiers spéciaux et « bigarrays »			
Primitive	**Paramètres**	**Arguments**	**Description**
`Paddbint`	type d'entier	`t1` et `t2` entiers boxés	addition `t1` + `t2`
`Psubbint`	type d'entier	`t1` et `t2` entiers boxés	soustraction `t1` − `t2`
`Pnegbint`	type d'entier	`t` entier boxé	opposé − `t`
`Pmulbint`	type d'entier	`t1` et `t2` entiers boxés	multiplication `t1` × `t2`
`Pdivbint`	type d'entier	`t1` et `t2` entiers boxés	division entière `t1` / `t2`
`Pmodbint`	type d'entier	`t1` et `t2` entiers boxés	modulo `t1` % `t2`
`Pandbint`	type d'entier	`t1` et `t2` entiers boxés	ET binaire `t1` && `t2`
`Porbint`	type d'entier	`t1` et `t2` entiers boxés	OU binaire `t1` || `t2`
`Pxorbint`	type d'entier	`t1` et `t2` entiers boxés	OU exclusif binaire `t1` ˆ `t2`
`Plslbint`	type d'entier	`t1` entier boxé, `t2` entier	décalage logique à gauche `t1` << `t2`
`Plsrbint`	type d'entier	`t1` entier boxé, `t2` entier	décalage logique à droite `t1` >>> `t2`
`Pasrbint`	type d'entier	`t1` entier boxé, `t2` entier	décalage arithmétique à droite `t1` >> `t2`
`Pbintcomp`	type d'entier C comparaison	`t1` et `t2` entiers boxés	comparaison `t1` C `t2`
`Pbintofint`	type d'entier	`t` entier	conversion de `t` vers un entier boxé
`Pintofbint`	type d'entier	`t` entier boxé	conversion de `t` vers entier
`Pcvtbint`	2 types d'entier	`t` entier boxé	conversion de `t` entre 2 entiers boxés
`Pbigarrayref`	nb de dimensions type de tableau et agencement	`t` tableau et `t_1`...`t_n` entiers	lecture du tableau suivant plusieurs dimensions
`Pbigarrayset`	nb de dimensions type de tableau et agencement	`t` tableau et `t_1` ...`t_n` couples d'entiers et de termes	écriture du tableau suivant plusieurs dimensions

Notes :

- Les primitives d'entiers spéciaux gèrent les types entiers boxés `Int32`, `Int64` et `Nativeint` (comme spécifié par leurs paramètres).
- Les gros tableaux sont multidimensionnels et se déclinent suivant les types suivants : quelconque, de flottants simple ou double précision, d'entiers 8 bits ou 16 bits signés ou non signés, d'entiers Caml, d'entiers 32 bits, 64 bits ou natifs, ou encore de nombres complexes sur 32 ou 64 bits. Ils peuvent avoir plusieurs agencements en mémoire, selon les conventions de C ou de Fortran.

Divers			
Primitive	**Paramètres**	**Arguments**	**Description**
`Pidentity`		un terme quelconque	retourne la valeur du terme
`Pignore`		un terme quelconque	jette la valeur du terme
`Pgetglobal`	identificateur id		lit la valeur de id dans les variables globales des modules
`Psetglobal`	identificateur id	un terme `t` entier	écrit la valeur de id dans les variables globales des modules
`Pbittest`		`t1` un bloc et `t2` un entier	voir la note ci-après

Notes :

- Les primitives `Pidentity` et `Pignore` ne sont pas stériles car le terme argument est évalué.
- `Pbittest` effectue un test de bit sur le bloc `t1` : les 3 bits de poids faible de l'entier `t2` déterminent un numéro de bit b de 0 à 7 et les bits de poids fort un offset k et on teste le bit b de l'octet lu au pointeur `t1`$+k$.

Ajout pour les besoins de `CtypedLambda`

Les primitives de `CtypedLambda` sont identiques à celles de `Lambda` et `Clambda`, à une exception près : la primitive `Pfield` se voit ajouter une version enrichie par un paramètre supplémentaire.

Primitive	**Paramètres**	**Arguments**	**Description**
`Pfldtag`	entiers i et tag	`t` un bloc (un variant de tag tag)	lecture `t`$[i]$

Primitives ajoutées dans le langage `ILM`

Lors du passage de `Ctypedlambda` à `ILM`, de nouvelles primitives sont introduites. Les principales sont illustrées sur le tableau suivant :

Primitive	**Paramètres**	**Description**
	Accesseurs	
`TPget_global`	nom d'un module	retourne le bloc global d'un module d'implantation
`TPget_field`	champ `fild_tid`	lit un champ de l'objet sur la pile
`TPset_field`	champ `fild_tid`	écrit un champ de l'objet sur la pile
`TPget_block`	entier	lit un champ d'un bloc générique sur la pile
`TPset_block`	entier	écrit un champ d'un bloc générique sur la pile
	Création d'objets	
`TPmktop`	référence `iltyperef`	crée une instance de fermeture sans environnement
`TPmkenv`	référence `iltyperef`	crée une instance de fermeture avec environnement
`TPmksclos`	référence `iltyperef`	crée une instance de fermeture partagée
`TPmkmrec`	référence `iltyperef`	crée une instance d'un membre de fermeture récursive
`TPbuildobject`	description de l'objet	crée une instance d'un objet quelconque
	Transtypage	
`TPcast`	types `ctype` t_1 et t_2	transtype la valeur sur la pile de t_1 à t_2
`TPconvint`	sorte d'entier boxé	convertit un entier quelconque en l'entier spécifié

Bibliographie

[1] Olivier ANDRIEU. « CamlOrbit ». oandrieu.nerim.net/ocaml/camlORBit, 2004.

[2] David BEAZLEY. « SWIG for CAML ». http://www.swig.org/Doc1.3/Ocaml.html.

[3] David BEAZLEY. « SWIG 1.1, Online Users Manual », 1997. www.swig.org.

[4] Nick BENTON et Andrew KENNEDY. « Interlanguage Working Without Tears: Blending SML with Java ». Dans *4th ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, 1999.

[5] Nick BENTON, Andrew KENNEDY et Claudio RUSSO. « The SML.net compiler ». www.cl.cam.ac.uk/Research/TSG/SMLNET/, 2003.

[6] Nick BENTON, Andrew KENNEDY, Claudio RUSSO et George RUSSELL. « SML.NET : Functional programming on the .NET CLR », 2003. http://www.cl.cam.ac.uk/Research/TSG/SMLNET/.

[7] Hans-Juergen BOEHM et Mark WEISER. « Garbage Collection in an Uncooperative Environment ». *Software — Practice and Experience*, 18(9):807–820, septembre 1988.

[8] Anne-Gwenn BOSSER et Francisco ALBERTI. « L'expérience Scol, un langage pour des applications internet multi-utilisateurs ». Dans *Journées Francophones des Langages Applicatifs (JFLA'03)*, février 2003.

[9] Sylvain BOULMÉ. « Modules, Objets et Calcul Formel ». Dans *Journées Francophones des Langages Applicatifs (JFLA'99)*. Inria, 1999.

[10] Gilad BRACHA, Martin ODERSKY, David STOURAMINE et Philip WADLER. « Making the future safe from the past: Adding Genericity to the Java Programming Language ». Dans *ACM SIGPLAN Conference on Object-Oriented Programming System, Languages and Applications (OOPSLA'98)*, octobre 1998.

[11] Yannis BRES, Bernard SERPETTE et Manuel SERRANO. « Bigloo.NET: compiling Scheme to .NET CLR ». *Journal of Object Technology*, 3(9):71–94, octobre 2004. Special Issue: .NET Technologies 2004 Workshop.

[12] Aravind C. « Understanding Classic COM Interoperability With .NET Applications », 2001. http://www.codeproject.com/dotnet/cominterop.asp.

[13] Nicolas CANNASSE. « CamlOLE ». http://tech.motion-twin.com/ocamole.

[14] Nicolas CANNASSE. « ODLL ». http://tech.motion-twin.com/odll.

[15] Clément CAPEL, Emmanuel CHAILLOUX et Jean-Marc EBER. « Application du toplevel embarqué d'Objective Caml ». Dans *Journées Francophones des Langages Applicatifs (JFLA'04)*, janvier 2004.

[16] Luca CARDELLI. « Compiling a Functional Language ». Dans *LISP and Functional Programming*, pages 208–217, 1984.

[17] Martin CARLISLE, Ricky SWARD et Jeff HUMPHRIES. « Ada for .NET, the A# homepage ». `www.usafa.af.mil/dfcs/bios/mcc_html/a_sharp.html`, 2004.

[18] Emmanuel CHAILLOUX. « An Efficient Way of Compiling ML to C ». Dans *Workshop on ML and its Applications*. ACM SIGPLAN, juin 1992.

[19] Emmanuel CHAILLOUX. « Dynamic Object Typing in Objective Caml ». Dans *International Lisp Conference*, 2002.

[20] Emmanuel CHAILLOUX et Grégoire HENRY. « O'Jacaré : une interface objet entre Objective Caml et Java ». Dans *Langages et Modèles à objets (LMO)*, mars 2004. `http://www.pps.jussieu.fr/~henry/ojacare`.

[21] Emmanuel CHAILLOUX, Pascal MANOURY et Bruno PAGANO. *Développement d'Applications avec Objective Caml*. O'Reilly, 1st édition, 2000. on-line english version : `http://caml.inria.fr`.

[22] Emmanuel CHAILLOUX, Raphaël MONTELATICI et Bruno PAGANO. « OCamIL : un compilateur Objective Caml pour .NET ». Dans Marc BUI, éditeur, *Proceedings of the Second International Conference RIVF'04 Research Informatics Vietnam-Francophony, Hanoï Vietnam*, volume Special issue of Studia Informatica Universalis, février 2004.

[23] Jérôme CHAILLOUX. « La machine LLM3 ». Rapport Technique, INRIA Rocquencourt, France, 1985.

[24] Jérôme CHAILLOUX, Matthieu DEVIN, Francis DUPONT, Jean-Marie HULLOT, Bernard SERPETTE et Jean VUILLEMIN. « Le lisp version 15.2, le manuel de référence ». Rapport Technique L-003, INRIA Rocquencourt, France, 1986.

[25] COM. « Component Object Model ». `msdn.microsoft.com/library/en-us/dnanchor/html/componentobjectmodelanchor.asp`.

[26] Jonathan COOK. « P#: a concurrent Prolog for .NET ». `http://homepages.inf.ed.ac.uk/stg/research/Psharp/`, 2003.

[27] Microsoft CORP. « .NET Framework Common Language Runtime Architecture ». supplied with VisualStudio.NET beta1 release, juin 2000.

[28] Guy COUSINEAU, Pierre-Louis CURIEN, Michel MAUNY et Gérard BERRY. « The categorical abstract machine ». *Science of computer programming*, 2(8):173–202, 1987.

[29] Vincent DANOS et Fabien TARISSAN. « Self-assembling Graphs ». *Natural Computing International Journal, Springer Verlag*, Mechanisms and formal tools in neural modeling: The analytic and synthetic approaches, 2006.

[30] Harley DAVIS, Pierre PARQUIER et Nitsan SÉNIAK. « Sweet harmony : The Talk/C++ connection ». Dans *Lisp and Functional Programming*, 1994.

[31] Daniel de RAUGLAUDRE. « camlp4 : Reference Manual ». Rapport Technique, Inria, 2002.

[32] Tyson DOWD, Fergus HENDERSON et Peter ROSS. « Compiling Mercury to the .NET Common Language Runtime ». Dans Nick BENTON et Andrew KENNEDY, éditeurs, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier, 2001.

[33] Edd Dumbill et Niel Bornstein. *Mono: A Developer's Notebook*. Developers' Notebooks. O'Reilly Edt, juillet 2004.

[34] ECMA International. « Standard ECMA-335. Common Language Infrastructure (CLI), Partitions I to V ». Rapport Technique, décembre 2002.

[35] Fabrice Le Fessant et Luc Maranget. « Optimizing Pattern Matching ». Dans *6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 26–37, 2001.

[36] Sigbjorn Finne. « Hugs98 for .NET homepage ». `galois.com/~sof/hugs98.net`, 2003.

[37] Sigbjorn Finne, Daan Leijen, Erik Meijer et Simon L. Peyton Jones. « H/Direct: A Binary Foreign Language Interface for Haskell ». Dans *3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, 1998.

[38] Sigbjorn Finne, Daan Leijen, Erik Meijer et Simon L. Peyton Jones. « Calling Hell From Heaven and Heaven From Hell ». Dans *4th ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, 1999.

[39] David Strickman Fox. « OCaml FFI Generator ». `ocamlffi.sourceforge.net`, 2002.

[40] Nicu G. Fruja et Egon Börger. « Analysis of the .NET CLR Exception Handling Mechanism ». Dans *.NET Technologies 2005 Workshop*, 2005.

[41] Richard P. Gabriel. « LISP: Good News, Bad News, How to Win Big ». *AI Expert*, 6(6):30–39, 1991.

[42] Adele Goldberg et David Robson. *Smalltalk-80, The Language and its Implementation*. Addison-Wesley, 1983.

[43] Rob Gordon. *Essential JNI: Java Native Interface*. Prentice Hall, 1998.

[44] K. John Gough. « Stacking Them Up: A Comparison of Virtual Machines ». Dans *Australasian Computer Systems Architecture Conference, Goldcoast, Queensland Australia*, pages 52–59. IEEE Computer Society Press, 2000.

[45] Pieter H. Hartel, Marc Feeley et *al.* « Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark ». *Journal of Functional Programming*, 6(4):621–655, juillet 1996.

[46] Jeff Henrikson. « Forklift ». `http://jhenrikson.org/forklift`.

[47] Grégoire Henry, Michel Mauny et Emmanuel Chailloux. « Typer la désérialisation sans sérialiser les types ». Dans *Journées Francophones des Langages Applicatifs (JFLA'06)*, janvier 2006.

[48] Richard W.M. Jones. « SimpleSoap ». `http://merjis.com/developers/simplesoap`.

[49] Richard W.M. Jones. « Perl4Caml ». `http://merjis.com/developers/perl4caml`, 2003.

[50] Andrew Kennedy et Don Syme. « Design and Implementation of Generics for the .NET Common Language Runtime ». Dans *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 1–12, juin 2001.

[51] Niels Kokholm et Peter Sestoft. « *Moscow ML .Net Internals* », 2003. online version : `http://www.dina.kvl.dk/~sestoft/mosml/netinternals.pdf`.

[52] Xavier LEROY. « The ZINC experiment : an economical implementation of the ML language ». Rapport Technique RT-0117, février 1990.

[53] Xavier LEROY. « *camljava* », 2000. `http://pauillac.inria.fr/~xleroy/software.html#camljava`.

[54] Xavier LEROY. « The Objective Caml system release 3.09 : Documentation and user's manual ». Rapport Technique, Inria, 2002. on-line version : `http://caml.inria.fr`.

[55] Xavier LEROY. « The CamlIDL homepage ». `http://caml.inria.fr/camlidl`, 2003.

[56] Erik MEIJER et Sigbjorn FINNE. « Lambada, Haskell as a better Java ». Dans *Proc. Haskell Workshop 2000*, 2000.

[57] Mercury TEAM. « The Mercury Project .NET ». `www.cs.mu.oz.au/research/mercury/dotnet.html`, 2003.

[58] MIDL. « Microsoft Interface Definition Language ». `msdn.microsoft.com/library/en-us/midl/midl/midl_start_page.asp`.

[59] Monique MONTEIRO, Mauro ARAÚJO, Rafael BORGES et André Luis SANTOS. « Compiling Non-strict Functional Languages for the .NET Platform ». *Journal of Universal Computer Science*, 11, 2005.

[60] Michal MOSKAL, Pawel W OLSZTA et Kamil SKALSKI. « Nemerle, Introduction to a functional .NET language ». `http://nemerle.org/intro.pdf`.

[61] Martin ODERSKY et Philip WADLER. « Pizza into Java: Translating Theory into Practice ». Dans *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[62] Francois PESSAUX et Xavier LEROY. « Type-Based Analysis of Uncaught Exceptions ». Dans *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 276–290, 1999.

[63] Pedro PINTO. « Dot-Scheme: A PLT Scheme FFI for the .NET framework ». Dans Matthew FLATT, éditeur, *Scheme Workshop*, pages 16–23, novembre 2003.

[64] François POTTIER et Didier RÉMY. The Essence of ML Type Inference. Dans Benjamin C. PIERCE, éditeur, *Advanced Topics in Types and Programming Languages*, Chapitre 10, pages 389–489. MIT Press, 2005.

[65] Steven PRATSCHNER. « Simplifying Deployment and Solving DLL Hell with the .NET Framework ». `msdn.microsoft.com/library/en-us/dndotnet/html/dplywithnet.asp`, 2001.

[66] Gnome PROJECT. « Orbit2 ». `http://www.gnome.org/projects/ORBit2`, 2002.

[67] PYCAML. « PyCaml ». `http://pycaml.sourceforge.net`.

[68] Christian QUEINNEC. *Lisp in Small Pieces*. Cambridge University Press, 1996.

[69] John R. ROSE et Hans MULLER. « Integrating the Scheme and C languages ». Dans *Lisp and Functional Programming*. ACM SIGPLAN, 1992.

[70] Francois ROUAIX et Jun FURUSE. « CamlTK ». `http://pauillac.inria.fr/camltk`.

[71] Didier RÉMY et Jérôme VOUILLON. « Objective ML: An Effective Object-Oriented Extension to ML ». *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[72] Thierry SAURA. « *Etude des modèles d'exécution des langages fonctionnels et impératifs Application à un évaluateur Scheme* ». PhD thesis, Université Paris 6, janvier 1999.

[73] Manuel SERRANO. « Bigloo homepage ». `www-sop.inria.fr/mimosa/fp/Bigloo`, 2004.

[74] Manuel SERRANO et Pierre WEIS. « Bigloo: a portable and optimizing compiler for strict functional languages ». Dans *2nd Static Analysis Symposium*, Lecture Notes on Computer Science, pages 366–381, Glasgow, Scotland, septembre 1995.

[75] Jon SIEGEL. *CORBA Fundamentals and Programming.* John Wiley & Sons, 1996.

[76] Julien SIGNOLES. « Defunctorisation at Work ». juin 2006. Unpublished.

[77] Raphael SIMON, Emmanuel STAPF et Bertrand MEYER. « Full Eiffel on the .NET Framework ». MSDN Library, `http://msdn2.microsoft.com/en-us/library/ms973898.aspx`, 2002.

[78] Raphael SIMON, Emmanuel STAPF, Bertrand MEYER et Christine MINGINS. « Eiffel on the Web: Integrating Eiffel Systems into the Microsoft .NET Framework ». `www.msdnaa.net/Resources/display.aspx?ResID=811`, 2000.

[79] Patrick SMACCHIA. *Pratique de .NET et C#.* O'Reilly, 1st édition, juin 2003.

[80] Gerd STOLPMANN. « OCaml FFI Generator ». `www.ocaml-programming.de/packages/fragments/cigen.tgz`.

[81] David STUTZ, Ted NEWARD et Geoff SHILLING. *Shared Source CLI.* O'Reilly Edt, mars 2003.

[82] Don SYME. « ILX: Extending the .NET Common IL for Functional Language Interoperability ». *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[83] Don SYME. « F# homepage ». `research.microsoft.com/projects/ilx/fsharp.aspx`, 2004.

[84] TEAMPLCLUB. « Winning entry of ICFP programming contest », 2000. Web page: `http://www.cis.upenn.edu/~sumii/icfp`.

[85] Thuan L. THAI et Hoang LAM. *.NET Framework Essentials.* O'Reilly Edt, 3rd édition, août 2003.

[86] W3C. « XSL Transformations (XSLT) ». `http://www.w3.org/TR/xslt`, 1999.

[87] W3C. « Document Object Model (DOM) Level 2 Core Specification ». `http://www.w3.org/TR/DOM-Level-2`, 2000.

[88] W3C. « XML Schema ». `http://www.w3.org/XML/Schema`, 2004.

[89] Philip WADLER. « Why No One Uses Functional Languages ». *SIGPLAN Notices*, 8:23, août 1998.

[90] Damien WATKINS, Mark HAMMOND et Brad ABRAMS. *Programming in the .NET Environment.* Addison Wesley, novembre 2002.

[91] X/OPEN GROUP. « *DCE 1.1: Remote Procedure Call* », 1994. `www.opengroup.org`.

[92] Dachuan Yu, Andrew Kennedy et Don Syme. « Formalization of Generics for the .NET Common Language Runtime ». Dans *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*. ACM SIGPLAN, janvier 2004.