PRATIQUE DES LANGAGES FONCTIONNELS TYPES

Emmanuel Chailloux

▶ **To cite this version:**

Emmanuel Chailloux. PRATIQUE DES LANGAGES FONCTIONNELS TYPES. Autre [cs.OH]. Université Pierre et Marie Curie - Paris VI, 2003. tel-00009013

HAL Id: tel-00009013
https://theses.hal.science/tel-00009013

Submitted on 13 Apr 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Habilitation à Diriger des Recherches

présentée à

L'UNIVERSITÉ PARIS VI

Spécialité

INFORMATIQUE

par

Emmanuel CHAILLOUX

Sujet du mémoire

Pratique des langages fonctionnels typés

Soutenue le 19 décembre 2003 devant la commission d'examen composée de

MM.	**Guy Cousineau**	Président
	Benjamin Goldberg	Rapporteurs
	Michel Mauny	
	Christian Queinnec	
	Gaétan Hains	Examinateurs
	Etienne Morel	

à *al-ma-ce* et *ma-ri-no*,
mes *passwords* préférés.

Avant-Propos

Ce mémoire présente une partie importante de mes activités de recherche depuis ma thèse. J'ai cherché à unifier leur présentation dans le cadre général des langages fonctionnels statiquement typés en dressant un panorama de leur pratique : de la compilation aux applications en passant par l'interopérabilité et les environnements de développement. Cela bien sûr sans oublier l'enseignement de ces langages, véritable moteur de diffusion.

L'introduction dégage les attentes vis-à-vis des langages algorithmiques séquentiels et la manière dont les langages fonctionnels statiquement typés, tout particulièrement le langage CAML, y répondent. Ce langage me sert d'articulation entre les différents chapitres. La présentation se veut plus intuitive que formelle, bien que certains passages soient assez techniques.

Les sujets abordés ont été découpés en six chapitres. Ils regroupent mes travaux anciens et récents liés à la programmation fonctionnelle typée. D'autres travaux sur la programmation par contraintes ou la programmation graphique et visuelle sont justes cités.

Les principales publications que ce mémoire réunit sont [32], [31] (issus de [30]), [81], [42], [28] et [39] pour les chapitres 1 et 3, [36], [62], [37] et [33] pour le chapitre 2, [40] pour le chapitre 4, [7], [41] et [124] pour le chapitre 5. Le chapitre 6 comprend des résultats issus des autres chapitres. La bibliographie est présentée en fin de document.

Il est toujours délicat de parler de travaux anciens comme s'ils venaient d'aboutir. J'ai essayé de reconstituer dans les introductions de chaque chapitre le cheminement de ces travaux et d'expliquer dans leur conclusion les orientations futures.

La conclusion du mémoire insiste sur l'importance du typage pour l'ensemble des thèmes abordés dans les différents chapitres en tenant compte des changements de la décennie passée. Les langages fonctionnels ont été influencés par les langages à objets et se sont ouverts au monde extérieur en interagissant avec d'autres langages et de nouvelles plate-formes d'exécution. Cette influence a été à double sens ; les langages à objets se durcissent au niveau des types. Qu'en sera-t-il demain ?

Par abus de langage, j'utilise le terme «langage fonctionnel typé» pour les langages statiquement/fortement typés. Bien sûr les langages fonctionnels dynamiquement typés garantissent à leur manière la vérification des types.

Remerciements

La plupart de mes travaux sont le résultat de collaborations. Certaines sont visibles comme les co-auteurs de publications : María-Virginia Aponte, Clément Capel, Jean-Marc Eber, Christian et Philippe Codognet, Guy Cousineau, Christian Foisy, Grégoire Henry, Pascal Manoury, Raphaël Montelatici, Bruno Pagano, Christian Queinnec et Ascánder Suárez.

D'autres collaborations sont moins perceptibles. Elles proviennent de groupes de travail, de discussions animées, de prototypes non finalisés, de spécifications en cours, etc. Je tiens à exprimer ma reconnaissance à l'ensemble des collègues qui ont participé à ces activités.

Merci à :

- Guy Cousineau, qui a initié et suivi mes travaux durant de longues années et qui me fait aujourd'hui le plaisir de présider ce jury,

- Benjamin Goldberg, Michel Mauny et Christian Queinnec qui ont accepté de rapporter ce mémoire. Chacun par son approche spécifique et ouverte des problèmes de typage a influencé mon travail. Leurs remarques ont été fort utiles à la finalisation de ce mémoire,

- Gaétan Hains, qui m'a fait goûté aux joies du parallélisme de données,

- Etienne Morel, pour son regard expérimenté sur les langages et les environnements de programmation.

Je remercie les différentes institutions qui m'ont accueillies, tout particulièrement :

- l'équipe PPS qui a réussi sur une courte période à construire des conditions de travail confortables et sympathiques tout en conservant le large spectre thématique de ses membres ;

- et l'ensemble de l'UFR d'informatique de l'université Paris 6 qui, par sa diversité, est source continuelle de discussions enrichissantes.

Je tiens à remercier la communauté du langage CAML qui a gardé son dynamisme au fil des ans grâce au projet Cristal de l'Inria qui année après année diffuse de nouvelles distributions de ce bien beau langage.

J'aimerai enfin remercier les cohortes d'étudiants qui ont chaque année, de manière différente, su renouveler mon plaisir d'enseigner.

J'ajouterai un dernier mot de remerciement à Pascal Manoury et Bruno Pagano, collègues et néanmoins amis, et qui le sont restés après la longue et difficile épreuve de l'écriture collective d'un livre.

Introduction

La programmation est une activité dont la complexité est bien reconnue tant du point de vue de spécialiste que de l'utilisateur des applications développées. Cette reconnaissance provient en partie de la richesse des logiciels utilisés mais aussi de leur fragilité à l'exécution. Il n'est pas rare de voir des comportements étranges d'utilisateurs dans leur emploi quotidien d'applications motivés par la peur que l'application interrompe brutalement leur travail en cours. Par contre l'activité de conception de langages ou d'environnements de développement n'apparaît pas primordiale en général, y compris pour l'étudiant en informatique. Une question survient alors «que peut-on chercher sur les langages ?», sous-entendu l'industrie du domaine veille à l'évolution des langages et de leurs utilisations. On peut y répondre en considérant cette autre question : «quelles sont les attentes vis-à-vis d'un langage de programmation». Nous répondions «la volonté d'abstraire» dans l'introduction de [41]. C'est ce fil directeur qui anime ce mémoire d'habilitation à diriger des recherches. Il porte sur la «*pratique des langages fonctionnels typés*». Ce titre englobe une thématique de conception et d'implantation des langages tout en s'intéressant aux outils facilitant leur emploi, aux démarches pédagogiques pour leur enseignement et à la diffusion de leurs applications.

Pour comprendre les choix qui ont dirigé nos travaux, nous présentons les caractéristiques désirées des langages de programmation et les critères de succès d'un langage. Ces éléments permettent d'apprécier le modèle de programmation fonctionnelle. Ce modèle qui apporte un haut niveau d'abstraction a été souvent critiqué. On analyse alors ces critiques anciennes et récentes qui restent un frein à l'utilisation de cette famille de langages. On présente ensuite une sélection de nos travaux qui répondent ou tentent de répondre aux critiques pertinentes. Ces recherches utilisent principalement le langage CAML comme cadre de travail. Ce langage, dont l'histoire est déjà ancienne, n'a connu qu'un succès d'estime et pourtant il séduit toujours de nombreux programmeurs.

Langages algorithmiques séquentiels

Les langages algorithmiques permettent de décrire des calculs sur des structures de données complexes. Ils sont à différencier des langages de format de données, des langages de scripts ou des langages de requêtes. On reste dans le cadre des langages séquentiels bien que la majorité des langages récents introduisent de la concurrence et un certain niveau de communication pour la répartition.

L'introduction de [41] explicite la volonté d'abstraction dans la conception et la genèse des langages.

Attentes vis-à-vis d'un langage

> « *Les langages de programmation sont en effet déjà nombreux et pourtant il en apparaît constamment de nouveaux. Au delà de leurs disparités, la conception et la genèse de chacun d'eux procèdent d'une motivation partagée : la volonté d'abstraire.*
> – *S'abstraire de la machine. En premier lieu, un langage de programmation permet de négliger l'aspect « mécanique » de l'ordinateur ; il autorise même à oublier le modèle du microprocesseur ou le système d'exploitation sur lequel sera exécuté le programme.*
> – *S'abstraire du modèle opératoire. La notion de fonction que possède sous une forme ou une autre la plupart des langages est empruntée aux mathématiques et non à l'électronique. D'une manière*

générale, les langages substituent des modèles formels aux conceptions purement calculatoires. Ils y gagnent en expressivité.
- *Abstraire les erreurs. Il s'agit ici de la tentative de garantir la sûreté d'exécution ; un programme ne doit pas se terminer brutalement ou devenir incohérent en cas d'erreur. Un des moyens pour y parvenir est le typage statique fort des programmes et la mise en œuvre d'un mécanisme d'exceptions.*
- *Abstraire les composants (i). Les langages de programmation donnent la possibilité de découper une application en différents composants logiciels, plus ou moins indépendants et autonomes. La modularité permet une structuration de plus haut niveau de l'ensemble d'une application complexe.*
- *Abstraire les composants (ii). L'existence d'unités de programmes a ouvert la possibilité de leur réutilisation dans d'autres contextes que ceux de leur développement. Les langages à objets constituent une autre approche de la réutilisabilité permettant la réalisation très rapide de prototypes.*

»

À cette liste on peut ajouter d'autres niveaux d'abstraction tant pour l'interopérabilité entre langages que pour la conception et la diffusion d'applications réalisées dans un langage :
- Abstraire les composants (iii) (persistance). La durée de vie d'un composant peut dépasser celle du programme qui l'a créé et peut être échangé avec d'autres programmes ne possédant pas les éléments pour le construire.
- Abstraire les composants (iv) (interopérabilité). Une application peut être vue comme un assemblage de briques logicielles provenant de plusieurs langages d'implantation. Les composants interopèrent à travers une interface de communication. Celle-ci permet de préparer les arguments et les résultats d'appel tout en gérant les allocations/récupérations mémoire entre plusieurs bibliothèques d'exécution.
- Abstraire la construction d'applications (portabilité). Une fois le programme construit, celui-ci peut s'exécuter sur différents couples processeur/système, ou à travers un navigateur sous forme d'applets. La notion de machine abstraite facilite cette diffusion. La bibliothèque d'exécution possède une interface avec le système en particulier au niveau de l'interface graphique. C'est une extension de l'abstraction de la machine.

Caractéristiques principales des langages séquentiels On peut découper les langages séquentiels en quatre grandes familles comme suit :
- modèle impératif : un programme est une suite d'instructions ; chaque instruction modifie un état, y compris le pointeur de code. C'est le **comment faire** où on explicite le contrôle du calcul.
- modèle fonctionnel : un programme devient le calcul d'une expression, l'enchaînement des calculs s'effectue par composition de fonctions. Une valeur fonctionnelle peut apparaître comme résultat de l'appel d'une autre fonction. C'est le **que faire** où l'ordre des calculs pour obtenir le résultat final importe peu.
- logique : à la différence du modèle précédent, l'unité de base est la relation. Un programme cherche donc à vérifier ou infirmer une relation, en construisant l'arbre de sous-buts à atteindre. En cas d'échec une autre branche est alors explorée. On reste dans le **monde déclaratif** où le contrôle n'est pas explicité.
- modèle objets : ce modèle privilégie les données par rapport aux traitements. Il offre une modélisation pour l'organisation des données. Les données sont alors encapsulées avec les traitements (méthodes) s'y référant. Les objets communiquent par appel de méthodes ; celles-ci pouvant modifier l'état interne. C'est un modèle **dirigé par les données**.

À ce premier découpage s'ajoute des caractéristiques principales sur les langages :
- typage : un langage peut être non typé, typé dynamiquement ou statiquement. Seuls les deux derniers cas peuvent être considérés comme sûrs à l'exécution. La différence entre eux vient du moment où la vérification de types intervient : à l'exécution ou à la compilation.

- polymorphisme : la possibilité de ne pas tenir compte de la forme (du type) des données pour les appels de fonctions ou de méthodes autorise un code plus général ; ce code est soit le même pour une famille de types de données (polymorphisme vertical), soit spécifique à chaque type (polymorphisme horizontal).
- structure : les niveaux de regroupement en classes pour les objets et en modules (ou paquetages) pour toutes les familles de langages facilitent plus ou moins le découpage logique d'une application dans un langage d'implantation.

Ces caractéristiques techniques cherchent en fait à répondre aux besoins «plus généraux» suivants :

- sûreté : éviter de se retrouver à l'exécution dans un état incohérent
- efficacité : l'exécution du programme est «satisfaisante»
- réutilisabilité : pouvoir reprendre dans une nouvelle application des parties déjà écrites auparavant sans avoir besoin de les modifier.

En règle générale on essaie de gagner sur ces trois facteurs.

Bien souvent un langage est un mélange des quatre familles précédentes. On peut citer par exemple :

- Mercury : logique et fonctionnel statiquement typé
- Ada 95 : impératif, objets, statiquement typé avec modules génériques
- Java : Objets, impératif, statiquement et dynamiquement typé, avec instance de classes locales reprenant la notion d'environnement des valeurs fonctionnelles.
- Objective Caml : fonctionnel, impératif, objets, typé statiquement avec modules paramétrés.

Genèse, influences et évolution d'un langage La naissance d'un nouveau langage provient le plus souvent d'un des éléments suivants :

- des besoins d'une application difficilement exprimables dans les langages existants ;
- d'un concept qui arrivant à maturité apporte un niveau d'abstraction novateur et pertinent
- d'une décision (technique ou marketing) de l'industrie
- d'une nécessité pédagogique
- du plaisir de ses concepteurs

Chaque nouveau langage porte en lui l'histoire des autres. L'influence d'un concept est d'une part due à ses qualités propres mais aussi au succès qu'il rencontre. Par exemple le paradigme objet se retrouve à toutes les sauces dans un très grand nombre de langages actuels, même si leur caractéristique principale n'est pas le modèle à objets. On peut citer l'évolution de C vers C++ et Objective C, l'intégration d'un modèle objet en Ada 95 ainsi que l'extension objet d'Objective Caml. Il en est de même des fonctions génériques à la Common Lisp que l'on retrouve maintenant dans certains Scheme comme Bigloo.

Viabilité et succès d'un langage Le nombre de créations de langages sur les cinquante dernières années est très important, de l'ordre de plusieurs milliers, mais seul un petit nombre de ces langages a connu une certaine notoriété.

Un langage tout d'abord doit exister, soit par une implantation, soit par une spécification. Le premier cas est le plus fréquent, néanmoins on rencontre au moins deux exemples du second cas : Standard ML et Ada. Pour Standard ML une sémantique formelle [107] a été donnée bien avant la première implantation complète. Ada a été sélectionné comme langage pour les application embarquées du *Dod*[1] par un appel d'offres sur spécifications [153]. Bien sûr les autres langages possèdent aussi une spécification, mais le plus souvent elle correspond à la spécification d'une implantation. Néanmoins une fois les premières implantations réalisées un effort de standardisation apparaît pour figer le noyau du langage.

Une fois un langage existant, celui-ci cherche à (sur)vivre. Pour cela il doit convaincre les futurs programmeurs, chefs de projets, enseignants, chercheurs de ses qualités tant au niveau des concepts qu'il véhicule que de la qualité de son implantation sans oublier le confort de son utilisation. Cette période de premiers pas est

[1] *United States Departement of Defense.*

assez cruciale pour atteindre un niveau de viabilité. Elle peut être soutenue par les qualités pédagogiques du langage, par un investissement marketing important d'une entreprise, par une application phare (*killer app*) ou résolvant élégamment un problème difficile, par un environnement facile à prendre en main, par un haut niveau de sûreté des applications produites, ...Une fois ce niveau de maturité/viabilité atteint un langage peut connaître différents niveaux de succès :

- les grands succès : Fortran, Cobol, C, C++, Visual Basic, Java
- les succès spécifiques : Sql, PostScript, Perl, PHP
- les succès du Web : Html, Xml, JavaScript
- des succès « niches » : Ada, Objective C, Erlang, Esterel
- des succès passés : Pascal
- des succès d'estime : SmallTalk, Prolog, Lisp, Scheme, ML, Eiffel, Haskell
- de possibles futurs succès : Python, C#
- des Ovni : Mercury, Oz,

On s'aperçoit dans cette liste qu'il y a des noms déjà ancien (presque cinquante ans) et d'autre plus récents. Néanmoins pour que le succès d'un langage puisse durer, celui-ci doit évoluer de manière importante tous les dix/quinze ans comme :

- Fortran IV, Fortran 77, Fortran 90
- Ada83 suivi d'Ada 95

ou même à un rythme plus rapide :

- Java 1.0, Java 1.1, ..., Java 1.4

en sachant qu'une évolution peut faire décliner le-dit langage en particulier quand celle-ci est décidée par une seule entreprise : Pascal suivi de Turbo Pascal ont rencontré un fort succès, mais le passage à l'objet Turbo Pascal Objet n'a pas été satisfaisant et il faudra attendre Delphi pour que Borland/Imprise renoue avec le succès. L'évolution est donc nécessaire mais aussi dangereuse, en particulier quand elle complexifie le langage initial dans la mesure où l'on tend à conserver une compatibilité ascendante et à intégrer de nouveaux traits pouvant être issus d'autres langages. Le prix du poids du passé, y compris en cas de succès, peut devenir élevé.

Quelle famille de langages ? : Nous cherchons à déterminer une famille de langages algorithmiques répondant de manière satisfaisante aux niveaux d'abstraction décrits précédemment. Le côté «algorithmique» élimine d'entrée de jeu le langage Prolog qui par un niveau d'abstraction trop élevé excelle dans la modélisation des relations mais ne facilite pas, sauf au prix de grandes contorsions, l'écriture d'algorithmes. La famille des langages impératifs ne répond pas au critère d'abstraction parce que leur modèle d'exécution reste proche de la machine et nécessite d'expliciter le contrôle d'exécution et la gestion mémoire. Restent la famille objet et la famille fonctionnelle. Là aussi le côté algorithmique détermine notre choix. Nous sommes à la recherche de langages qui calculent plutôt que de langages qui modélisent. Certes on peut écrire des algorithmes dans un style objet, mais quand on regarde comment implanter de manière généraliste des structures de données simples en objet on se heurte à une certaine verbosité qui n'aide pas leur écriture.

Niveau d'abstraction des langages fonctionnels Les langages fonctionnels répondent aux attentes d'abstraction, comme décrit page 1, tant du point de vue sûreté (gestion mémoire, typage et exceptions) que de la manipulation de composants. Glaser et Henderson concluent dans «Functional Programming and Software Engineering» [71] que «l'abstraction fonctionnelle est un outil puissant pour la décomposition d'un programme». Ce point de vue est illustré par des modèles fonctionnels de structuration. Hugues dans «Why Functional Programmimg Matters» [87] reprend cette argumentation au niveau de la programmation modulaire. Plus que la compilation séparée la programmation modulaire prend sa force dans la possibilité de composer des modules. La programmation fonctionnelle offre via les fonctions d'ordre supérieur un cadre puissant à la composition de modules. Norvig dans «A Retrospective on Paradigmes of AI Programming»[112]

dresse une liste de cinquante deux leçons pour le développement de programmes d'Intelligence Artificielle à laquelle les différents langages fonctionnels répondent.

Langages fonctionnels

Les langages fonctionnels peuvent être classifiés par les deux critères suivants :

1. typé statiquement ou dynamiquement
   Le premier critère indique quand le contrôle des types sera effectué. Si le typage est statique, ce contrôle a lieu au moment de la compilation du programme. Si le typage est dynamique, celui-ci est effectué pendant l'exécution du programme. Dans le cadre du typage statique, seuls les programmes corrects vis à vis de la vérification de types sont acceptés et pourront s'exécuter. L'intérêt est de garantir l'absence d'erreurs de types à l'exécution quitte à refuser des programmes justes (se terminant avec le bon résultat) ne respectant pas la discipline de types de l'algorithme de vérification de types. Dans le cadre du typage dynamique, tous les programmes syntaxiquement corrects, *i.e.* respectant la grammaire du langage, peuvent être exécutés, y compris des programmes faux du point de vue des types entraînant alors une rupture du calcul. On obtient dans le premier cas un cadre contraignant mais confortable et dans le deuxième cas une grande liberté avec ses dangers propres.
2. pur ou impur
   La caractéristique de pureté d'un langage fonctionnel est l'absence de contrôle explicite de l'exécution et de l'état mémoire du programme. Il n'y a pas de modification physique d'une zone mémoire (pas d'effets de bord). L'ordre de calcul des arguments d'une fonction ne modifie pas le résultat. Certaines structures de données classiques du monde impératif, comme les tableaux, ne peuvent se traduire directement.

   À l'inverse être impur autorise à manipuler l'environnement et effectuer des modifications physiques de valeurs. Cette caractéristique entraîne un choix pour la stratégie d'évaluation (des arguments d'une fonction) :
   - évaluation immédiate : les arguments des fonctions sont évalués dès leur passage dans le corps de la fonction. La conséquence est que, dès qu'il y a des effets de bord, l'évaluation doit être immédiate.
   - évaluation retardée : les arguments sont évalués dans le corps de la fonction soit une fois pour toutes à leur premier emploi, soit à chaque fois qu'ils sont utilisés.

Les langages fonctionnels purs implantent en général l'évaluation retardée. On obtient alors les trois familles suivantes :

- pur typé statiquement : Miranda, Clean, Haskell
- impur typé statiquement : ML (CAML et SML)
- impur typé dynamiquement : Lisp, Scheme, Erlang

Il y a peu d'exemples de langage pur typé dynamiquement. Hormis la tentative du Lisp impur paresseux HELP[2][134]. Néanmoins les langages impurs peuvent construire des données paresseuses comme décrit dans [2] pour Lisp ou dans [49] pour CAML.

Qui s'en sert ? Certes le titre de [154] «Why no one uses functional languages» est exagéré dans la mesure où :

- des langages déjà anciens comme Lisp/Scheme ou ML ont survécu aux différentes vagues comme C++ et Java, et ont su évoluer ; leurs communautés sont toujours vivaces comme le montrent les différentes listes de diffusions sur ces thématiques ;

[2]Help Est un Lisp Paresseux.

- des langages plus récents comme Haskell et aussi Mercury expriment leur dynamisme dans la communauté de recherche par le nombre d'articles aux conférences spécialisées ;
- des applications grand public ont été réalisées et sont quotidiennement utilisées ;
- des sociétés ont créé des outils et des applications commerciales,

et pourtant peu de programmeurs les citent comme leur langage de prédilection et l'ajout de ces connaissances dans un CV n'est pas forcément un plus pour les employeurs. Leur usage reste principalement confiné dans le monde académique : enseignement et recherche, quitte à déborder dans les centres de recherche privés. Cet état de fait doit être compris. Plusieurs documents ont, ces quinze dernières années, essayé d'analyser les principales critiques et de proposer des remèdes.

Critiques des langages fonctionnels Les critiques qui suivent reprennent et étendent les arguments de Gabriel [65]), de Wadler [154], de Serrano ([135]) ainsi que les répétitives discussions dans les listes de diffusion, en particulier celle de CAML[3].

sur l'efficacité : Une critique ancienne mais qui reste active porte sur les performances du point de vue rapidité d'exécution des langages fonctionnels. Cet argument date des premiers Lisp qui étaient interprétés et dont l'allocation et la récupération mémoire automatique n'étaient pas très performantes. On allouait trop, souvent inutilement, et on récupérait lentement. Pour cela la recherche sur l'implantation des langages fonctionnels a été importante et les résultats des vingt dernières années ont porté tant sur la qualité du code produit que sur les améliorations des techniques de GC[4]. Néanmoins cet argument est encore utilisé. Cela vient probablement du fait que les implantations de langages fonctionnels fournissent une boucle d'interaction (*toplevel*) pouvant faire penser au programmeur qu'il utilise un interprète. Or ces boucles d'interaction doivent seulement être considérées comme des outils de mise au point ou d'apprentissage. De plus le texte entré interactivement peut être compilé et chargé dynamiquement dans la boucle d'interaction. Un autre argument a été utilisé sur les techniques de compilation vers des machines abstraites. Bien que plusieurs implantations pouvaient soit produire du code-octet (*byte-code*) vers une machine abstraite avec expansion (appelée maintenant JIT pour *Just In Time*), soit du code natif vers une architecture processeur donnée, cet argument est resté présent pendant longtemps. Que peut-on dire de ces défauts de jeunesse ? L'apparition du langage Java les a en grande partie balayés. En fait ces défauts sont devenus des avantages grâce à Java. D'une part la gestion automatique de mémoire est maintenant considérée comme un gage de sûreté. D'autre part la portabilité due à la compilation vers du code-octet a été un des facteurs de succès de ce langage. Comme l'indiquait le marketing de SUN : «Compile once run everywhere». Du point de vue des performances, les premières versions de Java (sans JIT) ne brillaient pas par la vitesse d'exécution du code produit ce qui a permis des gains importants d'efficacité au fil des versions.

exécutables autonomes : Un deuxième argument ancien contre les langages fonctionnels provenait de la manière de créer des exécutables. Les techniques encore utilisées au début des années 80 créaient une image mémoire exécutable du programme chargé dans la boucle d'interaction. Cette image pouvait être gigantesque du point de vue mémoire car elle embarquait le compilateur (et l'interprète de code-octet) et contenait l'état du tas. De plus ces images n'étaient pas forcément autonomes : elles pouvaient avoir besoin de bibliothèques non encore chargées dans l'image. Soit elles étaient dépendantes d'une partie de l'environnement, soit elles ne fonctionnaient que sur une architecture donnée. D'autres langages comme SmallTalk utilisaient le même procédé. L'exécutable produit était soit très gros, soit non autonome. Gabriel dans [65] indiquait en 1991 qu'il était nécessaire de pouvoir construire de petits exécutables autonomes. Cela a été réalisé assez rapidement après [65] mais l'argument est encore utilisé.

[3] `caml.inria.fr/caml-list-eng.html`
[4] Garbage Collection ou Glaneur de Cellules.

interopérabilité : Un autre argument découle de ce dernier. Le monde des langages fonctionnels est quelquefois autiste (ou isolationniste). Il est difficile de les interfacer avec des bibliothèques existantes issues d'autres langages. Cela provient principalement de la gestion automatique de mémoire qui entraîne souvent des représentations spécifiques des valeurs. Là aussi cet argument commence à tomber en désuétude dans la mesure où il existe des interfaces avec C, et à travers C vers les autres langages. Dans certains cas, on rencontre des langages de description d'interface (*IDL*) qui facilitent cet interfaçage. Néanmoins cet argument n'est pas complètement tombé et se posent encore des difficultés entre langages possédant leurs propres gestionnaires de mémoire. La cohabitation n'est pas toujours de tout repos.

bibliothèques et portabilité : Une autre critique est liée à la précédente : les langages fonctionnels sont pauvres en bibliothèques, et ces dernières ne sont pas toujours portées sur les principales plate-formes de développement. Historiquement les langages fonctionnels étaient issus de la recherche académique. Une fois le langage spécifié et un prototype réalisé, il ne semblait pas que la construction de bibliothèques soit un thème de recherche. Cela a changé comme le montrent les distributions de Haskell, Objective Caml et Scheme qui possèdent de nombreuses bibliothèques. Néanmoins une partie de cette critique reste valable dans la mesure où celles-ci ne sont pas toujours *packagées* de manière cohérente et ne sont pas toujours portées sur les principales plate-formes de développement. Un exemple flagrant vient des interfaces utilisateurs 2D que l'on ne retrouve pas toujours sur les principaux systèmes de gestion de fenêtres : Windows, X-Window et Cocoa. Un langage comme Python a réussi dès sa sortie à fournir un grand nombre de bibliothèques y compris une interface 2D basée sur Tk et bien intégrée au langage.

difficultés d'apprentissage : Une critique récurrente sur les différents forums de discussion sur les langages fonctionnels ou de la part des étudiants en informatique vient de la difficulté d'apprentissage de tels langages. En fait cette critique revêt plusieurs aspects qui proviennent des points suivants :

- Le modèle de calcul sous-jacent (λ-calcul) est à la fois trop simple (seulement deux constructions) et trop complexe pour le codage de structures de données. Sa présentation est bien souvent trop mathématique.
- La syntaxe des langages fonctionnels est trop éloignée de celle de C ou Java, en particulier l'ancêtre Lisp de par sa syntaxe parenthésée.
- Le niveau d'abstraction est trop important. La compréhension de ce qu'est une valeur fonctionnelle n'est pas immédiate.
- Le typage statique et l'inférence de types sont difficiles à appréhender dans le cadre du polymorphisme paramétrique.
- Les constructions innovantes comme les continuations, les modules paramétrés, ou la macro-expansion sont des notions difficiles.
- La boucle d'interaction peut entraîner une certaine confusion sur l'interaction d'un utilisateur avec un programme.

On peut voir aussi la plupart de ces critiques comme des avantages. Néanmoins celles-ci doivent être prises en compte dans l'élaboration d'un enseignement dans la mesure où les notions difficiles abordées existent sous d'autres formes dans des langages qui semblent plus motiver les étudiants. La notion de valeur fonctionnelle n'est pas plus complexe que les instances de classe locale Java. Le système de types de C++ n'a rien d'évident. La syntaxe de HTML remplace des parenthèses par des balises.

environnement de développement : La plupart des distributions de langages fonctionnels offrent des outils simples comme le mode interactif (boucle d'interaction) et des modes d'édition structurée (principalement pour Emacs) et des outils plus évolués comme certains *debuggers* ou des profileurs exécution/mémoire. Les outils classiques du monde Unix peuvent aussi être utilisés comme la compilation conditionnelle (Makefile), gestion de version (CVS), ...Ces outils ne se retrouvent pas sur toutes les distributions, ni sur toutes les plate-formes. Il est à noter la faiblesse des outils pour Windows. Il existe de plus certaines difficultés venant du polymorphisme des langages fonctionnels typés et de leurs techniques de

compilation. En particulier l'absence d'informations de types à l'exécution ne facilite pas la tâche d'une trace ou d'un *debugger*. Les quelques rares essais d'environnements intégrés (*IDE*) n'ont pas été complètement probants. Comme pour la faiblesse du packaging de bibliothèque, le travail sur les environnements de programmation n'est pas toujours considéré comme un vecteur de recherche intéressant. Et pourtant l'environnement de développement est un élément important tant pour la prise en main d'un langage que pour le développement d'applications complexes. Le monde Lisp a été plus moteur d'une part parce que ce langage facilite l'usage de la réflexion et d'autre part par les produits industriels (ILOG VIEWS). Qu'il soit intégré ou non, un environnement de développement confortable pour la conduite de projets nous semble un élément indispensable pour le développement d'applications importantes et pour l'apprentissage des langages.

standardisation : Dans chaque famille de langages fonctionnels existe de nombreux dialectes. Historiquement on peut citer Common Lisp et Le_Lisp pour la famille Lisp et Standard ML et CAML pour la famille ML. Dans les deux cas leurs communautés respectives n'ont pas réussi à spécifier une norme, un standard pour ces langages. Par contre certains langages comme Scheme ont eu ce souci dès le début ce qui a abouti à une norme IEEE [44] qui de plus a su évoluer [90]. L'intérêt d'un standard officiel ou de fait est de permettre plusieurs implantations d'une même spécification et de garantir les évolutions en tenant compte de la compatibilité ascendante. Un standard tient compte de la communauté actuelle d'un langage et rend plus pérenne le développement dans celui-ci. Il favorise aussi l'investissement dans des produits commerciaux.

disponibilité et facilité d'installation : Bien que la compilation vers des machines abstraites soit une technique utilisée depuis des lustres par les compilateurs de langages fonctionnels, ceux-ci ne sont pas toujours disponibles sur une plate-forme spécifique. De plus les compilateurs natifs nécessitent une bonne connaissance des processeurs cibles. Mais dans le cas de disponibilité, l'installation n'est pas toujours aisée. Ces compilateurs sont le plus souvent issus du monde du logiciel libre. Une installation peut nécessiter plusieurs autres logiciels qu'il faut alors installer avant, cela que la distribution soit binaire ou source. D'autre part certaines bibliothèques ne sont pas portées sur certaines plate-formes, d'où des fonctionnalités amoindries.

commercialisation : Bien que les langages fonctionnels soient issus du monde académique plusieurs aventures industrielles ont essayé de les porter à un plus vaste public. Malheureusement ces expériences n'ont pas toujours eu le succès escompté pour différentes raisons. Les trois exemples suivants sont intéressants à étudier car les raisons de leur relatif échec sont différentes.

- Erlang a été conçu par la société Ericsson qui avait besoin d'un langage multi-processus pour la programmation des ses équipements téléphoniques. Plusieurs applications de téléphonie ont été conçues rapidement. Il a même débordé le cadre propre de la société. Néanmoins la société Ericsson a du suivre la vague Java pour intégrer les «standards» du marché bien que sa solution semblât plus adéquate pour ses besoins techniques. Ce langage a été mis dans le «domaine public»[5] par Ericsson. Cela lui a permis de survivre.
- Le_Lisp issu de l'Inria a été commercialisé par la société Ilog durant plusieurs années. Il a évolué et a été renommé `talk` pour masquer ses origines puis a été remplacé là aussi par C++ puis Java toujours pour satisfaire aux standards du marché. Il n'est plus guère utilisé car il est difficile de trouver ses distributions. Néanmoins la société Ilog existe toujours et est devenue une grosse PME (plusieurs centaines d'employés) de services.
- Scol langage de développement de mondes virtuels 3D de la société Cryo-networks s'était inspiré de Caml-Light. Il s'en différencie par son système de types. Il apportait un mécanisme simple de communication client-serveur et était muni de bibliothèques multimédia importantes pour la construction de scènes animées 3D. À la différence de VRML, langage de description de scènes, auquel on essaie d'ajouter des

[5] `www.erlang.org`

modules de communication, Scol est un langage de communication muni d'API multimédia. La société Cryo-networks a fermé durant l'été 2002. Le langage est maintenant *open source* [6].

La société Harlequin a commercialisé des environnements de développement pour Lisp et ML qui semblaient prometteurs. Elle a aussi fermé. Parmi ces sociétés pionnières on trouve encore Franz Inc[7] avec Allegro Common Lisp qui reste active en commercialisant des environnements de développement Common Lisp.

Certes ce survol des critiques, en particulier sur l'aspect commercialisation, peut sembler pessimiste pour le futur de ce modèle de programmation. Mais le fait que celui-ci ait survécu en évoluant indique que ses bonnes propriétés sont plus importantes que les points négatifs précédemment soulevés.

Quel langage fonctionnel ? : Il y a principalement le choix entre trois familles : impur typé dynamiquement, impur typé statiquement et pur typé statiquement. Le cadre confortable du typage statique nous paraît être un élément important tant par la plus grande sûreté des programmes exécutés que par la facilité d'interfaçage (bibliothèques, interopérabilité, ...) et la lisibilité du code. Toujours dans un souci de pragmatisme le côté élégant des langages fonctionnels purs nous semble un désavantage pour l'écriture d'algorithmes classiques, en particulier certaines structures de données impératives deviennent lourdes à manipuler dans ce cadre. On opte alors pour la famille des langages fonctionnels impurs statiquement typés dont un représentant est ML, tout particulièrement la branche des dialectes CAML.

Langage CAML

On précise par la suite les éléments de choix du langage CAML en retraçant sa déjà longue histoire et en analysant son utilisation actuelle.

Historique des langages à la ML ML est apparu initialement comme le *meta langage*, d'où son nom, de l'assistant de preuves LCF [73]. L'assistant LCF utilisait Lisp et ML dans son implantation. ML s'affranchit de LCF au milieu des années quatre-vingts en deux branches : Standard ML (Standard ML) et CAML (Categorical Abstract Machine Language). Ce clivage initial n'est pas sans rappeler les développements séparés de Lisp, en particulier les branches Common Lisp et Le_Lisp. Standard ML correspond à la branche anglo-saxonne (Edinhburg, Bell labs, Princeton). Ses premiers pas commencent par une spécification formelle du langage (noyaux et modules) [107] puis apparaissent plusieurs implantations dont la plus fameuse reste SML/NJ [10]. L'histoire de CAML [46], dont les principales implantations proviennent de l'INRIA, se découpe en trois périodes :

- CAML [156] (85-92) apporte la première implantation complète et portable de ML. Son nom vient de sa machine abstraite : *Categorical Abstract Machine*. Son implantation reposait sur le compilateur Le_Lisp, sa machine abstraite LLM3 et son expanseur de byte-code (JIT).
- Caml-Light [97] (90-97) était une implantation légère (*bootstrapable*) du noyau CAML utilisant une nouvelle machine virtuelle (ZINC), un GC efficace et des optimisations pour les applications totales.
- Objective Caml [100] (96-) améliore l'interprète de *byte-code*, offre un compilateur natif pour les principales architectures machines, introduit les modules paramétrés et les hiérarchies de classes pour l'organisation logicielle, ajoute la programmation multi-processus (*multi-threadée*).

D'autres compilateurs de CAML ont vu le jour pendant cette période, en particulier des compilateurs vers C au début des années quatre-vingt dix comme Camloo[136], Camelot [50] et CeML [32].

Aujourd'hui Objective Caml est un langage riche et complexe dont le noyau fonctionnel/impératif reste facilement abordable.

[6] `www.scol-technologies.org`
[7] `www.franz.com`

Utilisation de CAML CAML est principalement utilisé dans le monde académique : enseignement et recherche. Le recensement effectué par Guy Cousineau [47] en 1996 montrait son emploi dans les universités, écoles préparatoires et grandes écoles en France. Bien qu'il n'y ait pas d'étude comparable actuelle, le nombre de cours et le nombre d'étudiants ont probablement augmenté de manière significative.

ML reste un vecteur de recherche tant sur la partie langage (système de types, analyses statiques, modules, compilation) que comme laboratoire expérimental pour l'implantation de systèmes de preuves ou sur la mobilité du code. On peut citer comme réalisation le compilateur Objective Caml lui-même, les assistants de preuves Coq, Propre, Phox, PAF !, CAML déborde d'ailleurs le cadre purement académique pour intégrer des centres de recherche d'entreprises institutionnelles (CEA, FT, Dassault, EDF, ...).

Depuis peu des applications «grand public» réalisées en Objective Caml voient le jour et connaissent un certain succès comme `unison` pour la synchronisation de disques distants, `hevea` un traducteur LaTeX vers Html et `Active-DVI` pour conception de présentations animées. On rencontre même des articles dans la presse grand public qui décrivent ces applications (comme `unison` [102]) sans parler du langage d'implantation (alors qu'un article sur Objective Caml apparaissait dans le numéro du mois précédent [117]), ce qui est bon signe.

La conférence ICFP (*International Conference on Functional Programming*) organise depuis cinq ans une compétition annuelle de programmation[8] sur une durée limitée de trois jours où le langage d'implantation est libre. Chaque année une équipe utilisant Objective Caml est primée. Cela ne veut pas forcément dire qu'Objective Caml est un bon langage, mais seulement que de bons programmeurs savent l'utiliser et l'apprécier.

Par contre on ne rencontre que très peu d'applications commerciales qui utilisent CAML et qui s'en vantent comme par exemple les outils `mlfi`[9] pour les applications financières. Il existe probablement des applications internes aux entreprises développées dans ce langage mais elles restent discrètes. L'argumentation de Wadler [154] sur la prise de risque de choisir un langage en dehors des standards du marché justifie en grande partie cette frilosité. La présentation de Joe Amrstrong [13] expliquant comment convaincre son «boss» d'utiliser un langage fonctionnel montre une expérience avec Erlang et peut être lue avant d'aborder ce sujet.

Présentation des travaux

Je cherche dans la présentation d'une sélection de mes travaux à répondre de manière pertinente à la longue liste de critiques formulées précédemment dans le but de rendre la programmation fonctionnelle typée plus facilement praticable par un nombre plus important de programmeurs.

Chaque paragraphe suivant correspond à un chapitre du mémoire.

Compilation On compare les techniques de compilation de ML sur deux travaux : CeML travail réalisé dans la thèse de l'auteur [30] qui compilait ML vers C et un travail commun plus récent de compilation d'Objective Caml vers la plate-forme .NET. Les soucis de portabilité et d'interopérabilité étaient déjà présents à l'époque de CeML dans la mesure où l'on présentait une compilation efficace vers C vu alors comme un assembleur de haut niveau. Il est intéressant de mesurer l'augmentation de performances due d'une part à la rapidité des processeurs récents et d'autre part aux évolutions de la compilation de C. Bien sûr la production de fichiers intermédiaires C permet de construire des exécutables autonomes. La portabilité est grandement garantie par C quitte à nécessiter une nouvelle compilation. Par contre CeML proposait un système de modules différent de son ancêtre CAML. Cette non-standardisation n'a fait que s'accentuer avec l'apparition de Caml-Light et Objective Caml qui est de fait la référence actuelle pour la famille CAML. Pour concilier les propriétés de portabilité et d'interopérabilité avec celle de standardisation, nous avons entrepris avec Raphaël Montelatici la réalisation d'un compilateur Objective Caml pour la plate-forme .NET. Nous avons repris le travail commencé par Bruno Pagano sur la compilation de ML vers .NET. On présente ici

[8] `www.acm.org/sigplan/icfp.htm`
[9] `www.lexifi.com`

une première version encore naïve du compilateur O'CAMiL, compatible Objective Caml, produisant des exécutables .NET. Pour assurer cette compatibilité O'CAMiL développe, sous forme d'un *patch*, une nouvelle branche du compilateur Objective Caml pour cette plate-forme. L'approche est donc très différente de CeML. On cherche à être compatible avec Objective Caml et de faciliter l'interopérabilité du langage quitte à perdre en efficacité.

Extensions et compilation vers ML On montre ici les facilités d'extensions du langage Objective Caml par transformations de programmes ainsi que l'intérêt de choisir Objective Caml comme langage cible d'un compilateur. Pour cela trois travaux sont exposés.

Le premier travail est un compilateur du noyau du langage Scol. Il a été réalisé dans le cadre du projet EDICA[10] pour l'extension du langage Scol. Ce prototype a permis d'ajouter un mécanisme d'exception, les déclarations locales de fonctions, les types paramétrés et un système de modules. Ce compilateur transforme un programme Scol en programme Objective Caml.

Les deux travaux suivants concernent des extensions du langage. La première rétablit la notion de hiérarchie de classes importante pour la programmation objet classique et autorise le *downcasting* sous certaines conditions. Cette extension, appelée `coca-ml`, est ensuite utilisée pour ajouter la persistance objet au langage. La deuxième extension concerne la programmation fonctionnelle data-parallèle. Elle repart des travaux effectués dans cette voie avec Christian Foisy sur le compilateur `camlflight` pour en proposer une nouvelle implantation plus simple et profitant des nouveaux traits de programmation introduits en CAML comme les *threads*, la couche réseau et la persistance de fermetures.

Dans ces trois cas la richesse des outils de compilation d'Objective Caml est une aide appréciable, tout particulièrement `camlp4` qui est à chaque fois utilisé.

Interopérabilité Bien que la portabilité de CeML soit en grande partie garantie par le langage cible C, son interopérabilité avec des bibliothèques C nécessite une interface de fonctions externes (*foreign function interface* ou *FFI*). L'interfaçage de ML avec C lui fait perdre plusieurs caractéristiques de sûreté en particulier au niveau de la gestion mémoire et du partage/copie de valeurs. L'interface avec C d'Objective Caml rencontre les mêmes difficultés. Cela reste néanmoins fort pratique pour intégrer des bibliothèques C au langage CAML. On présente ici deux travaux récents. Le premier est l'embarquement sous forme de bibliothèque du *toplevel* Objective Caml. Ce travail a été réalisé avec Clément Capel et Jean-Marc Eber. Cet embarquement autorise la compilation et l'évaluation de code Objective Caml dans une application C ; celle-ci lui passant le source à évaluer. Cela facilite l'écriture de *plugin* et de langage de script. Le deuxième travail cherche à faire communiquer de façon sûre, tant du point de vue du typage que de la gestion mémoire, Objective Caml avec Java. Pour cela nous avons défini avec Grégoire Henry un *IDL* (*Interface Description Language*) entre les modèles objets des deux langages. À partir d'un fichier de description de classes, un traducteur construit les classes englobantes et autorise les *callback* entre les deux langages. On arrive ainsi à construire une interface graphique Java pour un programme Objective Caml sachant que les objets traitant les événements sont définis en Objective Caml.

Environnement de programmation Comme signalé précédemment, les recherches sur les environnements de programmation pour les langages fonctionnels typés statiquement polymorphes paramétriques ne sont pas légion. Les travaux que nous présentons reflètent cet état. On se contente de montrer deux voies pour la construction d'un environnement complet. Le premier travail montre l'utilisation de l'outil `eclipse` pour la construction d'un *IDE* (*Integrated Development Environment*) pour Objective Caml. Ce travail récent principalement réalisé par Alexandre Deckner et Frédéric Dallot sous ma direction construit un environnent intégré pour Objective Caml, appelé Ocaide, introduisant la notion de projet, l'utilisation d'un éditeur structuré en liaison avec la compilation, la navigation dans les sources et intègre des outils disparates comme les

[10] Environnement de Développement Interactif (grand public) pour la Conception d'Agents dans des mondes virtuels.

dépendances de compilation et la gestion de version. Le deuxième travail présenté s'intéresse à la reconstruction dynamique de types pour ML. L'information de typage issue de la compilation ne suffit pas dans le cadre du polymorphisme paramétrique à déterminer exactement les types des valeurs à l'exécution. En effet il manque une information liée à l'histoire du calcul tant pour les effets de bord, les ruptures de calcul et les applications partielles. Cette information est nécessaire pour les outils de mise au point, pour une persistance sûre et pour effectuer des contraintes de types en objet. On définit alors une reconstruction basée sur une information de types liée aux valeurs. Pour diminuer son coût on propose de n'utiliser qu'une information succincte (les types superficiels) permettant de reconstruire une approximation du type attendu. On présente un nouvel algorithme de GC, dérivant d'un `Stop&Copy` utilisant la mémoire non utilisée pour ranger les types superficiels.

Enseignements Notre expérience de l'enseignement de CAML sur les dix dernières années permet d'envisager une meilleure diffusion et compréhension du langage si l'on sait adapter cet enseignement aux différents publics. ML comme premier langage informatique à un public d'étudiant ne se présente pas comme aux futurs MASTER de recherche. Entre ces deux extrêmes il y a plusieurs déclinaisons de son apprentissage soit orienté «types et structures de données», soit comme langage multi-paradigmes manipulant objets et modules paramétrés. ML peut aussi servir comme langage support à de nombreux cours tant d'algorithmique, de compilation, de systèmes et réseaux, etc. Il nous paraît être aussi un choix intéressant pour d'autres disciplines scientifiques, tout particulièrement en mathématiques, physique et linguistique. Bien sûr il peut être prodigué en formation permanente pour un public ingénieur. Enfin il peut faire l'objet d'auto-apprentissage en s'intégrant dans une formation utilisant les TICE[11].

Diffusion d'applications La facilité de diffusion d'applications est un des éléments de succès d'un langage. On désire d'une part pouvoir écrire des applications portables mais aussi de pouvoir les diffuser pour des systèmes spécifiques. Pour cela on présente une utilisation du *toplevel* embarqué pour l'exécution d'application à l'intérieur d'un navigateur Web. Cette forme de diffusion a été l'un des vecteurs de succès du langage Java. L'interfaçage avec Java permet aussi d'obtenir un bon niveau de portabilité dans la mesure où Java fonctionne sur les principaux systèmes. Mais dans le même temps on désire pouvoir facilement installer une application sur un système particulier. On montre alors en utilisant le compilateur O'CAMiL la construction d'applications directement exécutables sous .NET. Dans ces trois cas l'utilisateur final n'a pas à se soucier d'installation. La tâche du concepteur d'applications en est aussi simplifiée.

[11] Technologie de l'Information et de la Communication dans l'Enseignement.

Chapitre 1

Compilation de ML

La compilation de ML se place tout naturellement dans le cadre de la compilation des langages fonctionnels qui nécessite :

- d'expliciter le contrôle (application)
- d'expliciter la gestion de l'environnement (fermetures).

On retrouve ces difficultés dans d'autres langages fonctionnels comme Haskell ou Lisp/Scheme. Certaines techniques de transformation de programmes permettent d'expliciter la gestion de l'environnement (λ-lifting [88]) et d'expliciter le contrôle de l'exécution (*Continuation Passing Style* (CPS) [9]). Elles sont décrites dans le chapitre 2 de la thèse de l'auteur [30]. De plus comme ML est un langage fonctionel impur, la partie non-fonctionnelle du langage doit être traitée avec autant d'attention. Ce point est rappelé dans la «brève histoire de Caml» de Guy Cousineau[1].

Plusieurs compilateurs de ML ont vu le jour depuis le milieu des années 80, pour atteindre aujourd'hui d'excellents niveaux de performance. Les principaux efforts ont porté sur les fermetures : représentations optimisées, analyses pour éviter d'allouer inutilement de telles valeurs, et sur les algorithmes de GC. Du point de vue gestion mémoire cela se répercute en allouant moins souvent, en créant des objets plus petits et en les récupérant plus vite. Bien sûr les langages cibles ont aussi eu leur importance. On peut diviser en trois voies ces travaux : vers des machines abstraites, vers des assembleurs portables de haut niveau comme C, vers de réels assembleurs de bas niveau. Ainsi les modèles de machines abstraites privilégiant initialement le partage de l'environnement comme la CAM [48] pour réduire la taille des fermetures ont évolué pour optimiser l'accès à l'environnement comme la FAM [29] en copiant alors l'environnement. Ces deux modèles se retrouvent pour des langages cible comme C ou des assembleurs de machines réelles.

La première section rappelle rapidement les principaux choix techniques pour la compilation de ML en insistant sur l'importance de la bibliothèque d'exécution pour apprécier ceux effectués par les deux compilateurs présentés ensuite.

La deuxième section actualise le travail sur CeML, compilateur de ML vers C, en montrant la portabilité de cette technique et les gains de performance obtenus sur les nouvelles architectures machines.

La troisième section présente le compilateur O'CAMiL[2][108], basé sur le compilateur Objective Caml et compatible avec lui, pour la plate-forme .NET. Son but est de mesurer les facilités d'interopérabilité promises par .NET. Ce travail, initié par Bruno Pagano, est commun à Raphaël Montelatici et l'auteur. On ne présente ici que la première version du compilateur ; ses optimisations et l'interopérabilité avec d'autres langages seront discutés dans la thèse de Raphaël Montelatici.

La dernière section discute de l'importance de l'information de types lors de la compilation.

[1] `www.pps.jussieu.fr/~cousinea/Caml/caml_history.html`
[2] `www.pps.jussieu.fr/~montela/ocamil`

1.1 Techniques de compilation

La famille des langages ML possède certaines caractéristiques qui influencent les techniques de compilation, en particulier les trois points suivants :
- le polymorphisme paramétrique,
- l'application partielle
- et le filtrage de motifs.

Les autres points abordés dans ce chapitre sont plus classiques et sont liés à la bibliothèque d'exécution (*run-time library*), la génération de code et l'édition de liens. La bibliothèque d'exécution contient le gestionnaire de mémoire (GC) qui est lié à la représentation des valeurs. La génération de code diffère fortement selon le niveau du langage cible. Enfin l'édition de liens nécessite des informations, principalement de typage, sur les déclarations exportées.

Représentations des données On distingue principalement deux classes de données :
- les valeurs immédiates qui tiennent dans l'espace d'un mot mémoire (entiers, ...)
- les valeurs structurées plus grosses qu'un mot :
  1. allouées dans le tas ; l'accès à ces valeurs s'effectue en suivant son pointeur d'adresse ;
  2. allouées dans la pile ; en prenant le nombre de mots nécessaires.

Ce qui peut se résumer dans le premier cas en «entiers ou pointeurs» et dans le second cas en «entiers ou blocs entiers».

De par le polymorphisme paramétrique les deux appels suivants de la fonction `app` doivent pouvoir être traités :

map version 1
```
# let rec map f l = match l with
  [] -> []
  | h::q -> let nh = f h in nh :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let app f x = f x;;
val app : ('a -> 'b) -> 'a -> 'b = <fun>
# app succ 3;;
- : int = 4
# app (map succ) [1;2;3];;
- : int list = [2; 3; 4]
```

La fonction `app` attend une valeur fonctionnelle `f` et un argument `x` sur lequel sera appliqué l'argument `f`.

Il y a principalement deux techniques[109] pour la représentation des valeurs structurées :
- représentation uniforme : le type α (`'a`) correspond à un mot machine. Si une valeur est trop grande pour rentrer dans un mot, elle est alors allouée dans le tas et c'est son adresse qui est utilisée. On dit que la valeur est encapsulée (*boxed*).
- représentation *ad hoc* : une information de type (*tag*) est ajoutée aux valeurs. Cette information est ensuite utilisée à l'exécution pour l'aiguiller vers les instructions en dépendant.

Applications partielles et totales En λ-calcul on considère toute déclaration de fonction comme une fonction à un paramètre. Le lieur λ ne s'applique qu'à un identificateur. Comme le λ-calcul est le modèle sous-jacent de ML, on peut considérer toutes les fonctions comme des fonctions à un paramètre. Néanmoins un compilateur efficace de ML a grandement intérêt à introduire une notion d'arité des fonctions et de détecter leurs applications totales (quand tous les arguments attendus sont passés) pour améliorer le code produit. C'est d'ailleurs une des principales optimisations des compilateurs ML. Le premier compilateur CAML V2.6-1 n'effectuait pas cette optimisation ; la conséquence est de créer des valeurs fonctionnelles intermédiaires

inutiles ralentissant l'application et le GC. On retrouve cela plus récemment avec le modèle objet d'Objective Caml qui considère les méthodes comme des valeurs fonctionnelles et ne permet pas la détection statique d'une application totale.

Filtrage de motifs Le filtrage de motif est un assemblage syntaxique permettant de nommer ou de tester une valeur ou une partie d'une valeur structurée. Il s'intègre au système de types de ML. La fonction `map` de la page 14 définit un filtre simple pour les listes avec deux motifs : le premier motif teste si la liste est vide (`[]`), le second motif nomme pour une liste non vide sa tête (`h`) et sa queue (`t`).

Le filtrage de motif est l'unique moyen de déstructurer une valeur d'un type somme. Comme plusieurs structures de données de base comme les listes et les arbres sont construites à partir de tels types, le filtrage de motifs est une partie à optimiser pour un compilateur ML [119].

Les techniques actuelles permettent de construire un filtre optimal par rapport au nombre d'opérations effectuées comme dans [121, 94].

Bien que ne faisant pas partie de la compilation optimale du filtrage, un élément important est l'explicitation des «*warnings*» : filtrage incomplet ou clause non utilisée, comme le rappelle Luc Maranget dans [104].

Inlining L'*inlining* remplace le site d'appel d'une valeur fonctionnelle sur un argument par le code de cette fermeture où le paramètre formel est remplacé par l'argument passé. Il y a deux conséquences immédiates : l'exécution est accélérée et la taille du code peut augmenter.

Bibliothèque d'exécution La bibliothèque d'exécution d'une implantation de langages fonctionnels prend en charge :

- la gestion automatique de mémoire (GC),
- le mécanisme général d'application qui lors du passage d'un argument à une valeur fonctionnelle décide soit de la création d'une nouvelle fermeture, soit de l'exécution du code de la valeur fonctionnelle,
- la gestion des exceptions : pose d'un récupérateur, déclenchement et récupération
- l'interfaçage avec des fonctions système (entrées/sorties, ...)

On retrouve cette bibliothèque de bas niveau quelle que soit la technique de génération de code.

Génération de code La génération de code suit plusieurs phases passant de l'arbre de syntaxe décoré ou non par l'information de typage à une liste d'instructions du langage cible. Celui-ci peut être un langage de haut niveau (comme C) ou un langage de bas niveau comme l'assembleur d'une machine réelle ou une machine abstraite. Dans ce dernier cas les instructions seront évaluées par l'interprète de cette machine. Les phases intermédiaires de la génération de code varient selon la machine cible. Certaines optimisations comme l'allocation de registres prennent leur sens pour des machines réelles où ceux-ci sont une denrée rare. Ces différentes phases linéarisent de plus en plus l'arbre de syntaxe jusqu'à obtenir une liste d'instructions optimisée.

Édition de liens L'édition de liens relie les différents composants d'un programme avec la bibliothèque d'exécution et le système d'exploitation. Selon les techniques de génération de code un certain travail peut être nécessaire à l'édition. De plus certaines optimisations ne peuvent être effectuées qu'à ce moment là par des analyses du programme complet. Une partie de l'édition de liens peut être reportée au chargement du programme où chaque module exécute sa phase d'élaboration.

1.2 Compilation vers C : CeML

CeML est un traducteur d'un dialecte de ML, proche de Caml-Light, vers du C de haut niveau : une fonction ML correspondra à une fonction C. Il correspond au travail de thèse de l'auteur [30]. Il a évolué en 92/93 en intégrant un filtrage optimal. Depuis il n'a pas été modifié mais a juste été porté sur de nouvelles plates-formes comme Linux. On cherche à suivre l'évolution de ses performances qui ne sont pas seulement dûes à la rapidité des nouveaux processeurs, mais aussi aux optimisations apportées aux compilateurs C.

1.2.1 Description générale

CeML est un descendant de CAML V2.6-1 qui a été défini en 1989 et implanté en 1990/91. Du point de vue langage il diffère de son illustre ancêtre par un système de types qui conserve une information d'arité et par son système de modules simples. Il se différencie aussi par les techniques de compilation utilisées et sa bibliothèque d'exécution.

du C de haut niveau Le code produit est du C de haut niveau où une fonction ML est traduite dans une fonction C. Comme C ne possède pas de fonctions locales, toutes les fonctions locales CeML deviennent des fonctions globales par λ-lifting [88]. Les variables libres des fonctions locales sont vues alors comme des paramètres supplémentaires. Dans le cas de déclarations de fonctions mutuellement récursives, les identificateurs de ces fonctions sont elles aussi ajoutées comme paramètres. Il y a deux façons de passer l'environnement (l'ensemble des variables libres) d'une fonction locale globalisée. Soit on les ajoute sous forme d'une structure les regroupant (un enregistrement), soit chaque variable libre devient un paramètres supplémentaire. Le premier cas alloue une structure et nécessite une indirection pour l'accès à une telle variable. Le second cas n'alloue pas et effectue un accès direct, mais le nombre de paramètres peut devenir important empêchant alors certaines optimisations. C'est cette dernière technique qui a été choisie pour rendre le code engendré plus lisible.

représentation des données Toutes les données sont représentées de manière uniforme. Une valeur sera un mot mémoire, 32 bits sur les architecture 32 bits ... Les valeurs immédiates sont celles qui tiennent dans cet espace mémoire, comme les entiers, les constructeurs constants, les caractères et les nombre flottants à simple précision. Toutes les autres données sont vues comme un pointeur mémoire. Ces données sont allouées dynamiquement dans le tas de CeML. On définit un type α (`char *`) vers lequel tout type d'une valeur peut être contraint et réciproquement. Le type α d'une fonction polymorphe pourra alors accepter un entier, un flottant ou un pointeur.

valeurs fonctionnelles et applications Toutes les abstractions (`fun`) liées à un nom sont globalisées en CeML. Les abstractions non liées sont alors automatiquement nommées et traitées comme des déclarations. Les seules fonctions obtenues sont des fonctions globales sans environnement, toutes les variables apparaissant dans le corps d'une fonction sont soit des variables globales, soit des paramètres, soit des variables locales, soit des variables issues d'un motif. Par exemple si on définit `map` de la manière suivante :

map version 2

```
let map f l =
  let rec map_aux = function
    [] -> []
   | h::t -> let nh = f h in nh :: (map_aux t)
  in map_aux l;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

La fonction locale `map_aux` possède une variable libre : la fonction `f`. Par λ-lifting on globalise la fonction locale :

map version 3

```
let rec map_aux f = function
    [] -> []
   | h::t -> let nh = f h in nh :: (map_aux f t);;
val map_aux : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let map f l = map_aux f l ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Les valeurs fonctionnelles (ou fermetures) sont constituées d'un couple : (environnement, code) où le code est une de ces fonctions globales C et l'environnement un vecteur qui contient les paramètres déjà passés à cette fonction.

Pour diminuer le nombre de fonctions déclarées, on utilise une information d'arité sur ces fonctions. Pour la calculer on regroupe une suite d'abstractions : `function p1 -> function p2 -> ... -> function pn -> e` comme une fonction à n paramètres. Cette information est calculée au typage qui la propage (voir paragraphe suivant).

L'application partielle d'une fonction à n paramètres crée une fermeture. Les paramètres passés sont conservés dans l'environnement de la fermeture. L'appel suivant : `let am = map succ;;` construit une fermeture dont l'environnement contient l'argument fonctionnel `succ` et le code la fonction `map`.

typage et arité Pour conserver l'information d'arité d'une fonction, CeML introduit deux constructeurs de types fonctionnels. : `->` et `=>`. La simple flèche `->` est le constructeur fonctionnel habituel. La double flèche `=>` construit un type fonctionnel dont l'arité est précisée. Par exemple le type $\tau_1 => \ldots => \tau_n => \tau$ correspond à une fonction d'arité n.

L'introduction du constructeur de type `=>` intervient au filtrage muni de `function` en suivant la règle suivante :

$$\frac{C \vdash pat_i : \tau_i \quad expr : \tau}{C \vdash \texttt{function}\, pat_1 - > \ldots \texttt{function}\, pat_n - > expr : \tau_1 => \tau_2 => \ldots \tau_n => \tau}$$

Sa règle d'application totale est la suivante :

$$\frac{C \vdash expr : \tau_1 => \ldots => \tau_n => \tau \quad C \vdash atexpr_i : \tau_i (1 \leq i \leq n)}{C \vdash expr\ atexpr_1 \ldots atexpr_n : \tau}$$

mais comme ce cas n'est pas obligatoire, il est nécessaire d'avoir aussi une règle d'application partielle :

$$\frac{C \vdash expr : \tau_1 => \ldots => \tau_n => \tau \quad C \vdash atexpr_i : \tau_i \quad (1 \leq i \leq k \ \texttt{avec}\ k < n)}{C \vdash expr\ atexpr_1 \ldots atexpr_k : \tau_{k+1} \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau}$$

Il s'élimine ainsi dans le cas général :

$$\frac{C \vdash expr : \tau_1 => \tau_2}{C \vdash expr : \tau_1 \rightarrow \tau_2}$$

Cette différence entre constructeurs fonctionnels est invisible dans l'utilisation du langage. Par contre, elle est particulièrement utile pour l'optimisation des applications totales.

filtrage de motifs Le filtrage de motifs reprend la sémantique de CAML V2.6-1 en autorisant le filtrage multi-colonnes.

filtrage CAML V2.6-1

```
fun p_1_1 ... p_1_n -> e1
 |  p_2_1 ... p_2_n -> e2
...
 |  p_k_1 ... p_k_n -> ek
```

Son implantation construit un arbre optimal en suivant la technique décrite dans [121].

GC à racines ambiguës Les *Garbage Collector* sont des algorithmes explorateurs du tas (*heap*) qui ne conserveront que les valeurs allouées encore utiles au programme. Pour faciliter cette exploration les GC différencient en règle générale les valeurs immédiates des pointeurs. Cela permet de ne suivre que des vrais pointeurs.

Néanmoins l'étiquetage des valeurs immédiates a un coût : soit d'avoir une arithmétique spéciale, soit d'effectuer une opération à chaque pointeur suivi. De plus ce marquage ne permet pas d'utiliser l'intervalle de valeurs prévu pour les types de base comme les entiers. On rencontre souvent des entiers 31 ou 30 bits pour des mots de 32 bits. CeML se voulait proche de C. Pour cela il évite de marquer les valeurs immédiates et les pointeurs. Cela n'enlève rien à la sûreté d'exécution [8] dans la mesure où le typage statique garantit l'absence d'erreur de types à l'exécution. Cette architecture d'étiquetage n'est pas toujours nécessaire [15]. Certaines heuristiques peuvent être employées pour faire cette distinction entre valeurs immédiates et pointeurs dans tous ou un grand nombre de cas. Ces méthodes se justifient en garantissant qu'une mauvaise discrimination n'entraîne pas d'erreur et ne créent, selon les applications, qu'une utilisation de mémoire superflue. Cela n'est malheureusement pas suffisant. L'exploration d'un faux objet peut casser la structure de la mémoire. Certains algorithmes, adaptés à une organisation mémoire et à une application, peuvent néanmoins éliminer ces cas dangereux ([22, 15]) et mettre à jour les racines en cas de déplacement ([16]). Ce type de GC est appelé GC à racines ambiguës.

Les caractéristiques du GC à racines ambiguës de CeML sont :
- ensemble de racines (*root set*) géré par le *runtime* (n'utilise pas la pile C) ;
- GC conservatif : les objets alloués ne sont pas déplacés ;
- le tas est découpé en zones d'allocation d'objets de taille 2^n pour $n \leq 10$ auxquelles est ajouté une zone pour les objets de plus grande taille ;
- un `Mark&Sweep` est effectué pour chaque zone.

modules Un programme CeML est découpé en plusieurs fichiers. Chaque fichier est appelé unité de compilation. Chaque unité peut être compilée séparément des autres unités du programme. Chaque unité possède une partie «importation» où elle indique quelles sont ses dépendances par rapport aux autres unités, et une partie «exportation» où elle indique les déclarations globales qu'elle désire rendre accessibles aux autres unités. Les dépendances sont nommées explicitement à la différence des modules de Standard ML ([122]) et de CAML V2.6-1 ([156]).

1.2.2 Évolution des performances

On cherche à comparer les tests de performance de l'époque, qui ont été effectués sur des SUN3 (Motorola 68020) et des DecStation (MIPS R2000), par rapport à des architectures plus récentes des stations Linux (Pentium). L'idée n'est pas de décider quel est le compilateur ML le plus performant, mais d'apprécier l'évolution de puissance des compilateurs ML. Pour cela on rappelle tout d'abord les tests de comparaison effectués dans [30] pour ensuite les faire tourner sur des processeurs plus récents, puis d'ajouter des tests demandant plus de puissance. Pour simplifier l'écriture de ces tests, on utilise le même sous-langage pour les différents dialectes CAML sans utiliser leurs systèmes de modules qui diffèrent.

Anciens tests de performance

Ces tests sont regroupés en deux séries. La première série contient des tests simples qui essaient de mesurer une ou deux caractéristiques de l'implantation. On peut les regrouper ainsi :
- petites fonctions sur les entiers à un ou plusieurs paramètres : `Fib`, `Tak`, `Fiblt`, `Fibgt`
- petites fonctions sur les chaînes : `Makestring`, `CountStr`
- fonctionnelles sur les listes : `Rev`, `SigmaMap`, `ItList`
- λ-calcul pur : `Double`

– exceptions : `TakE`
– vecteurs : `SigmaVect`, `SigmaVectW`

Les figures 1.1 et 1.2 sont celles données dans [30]. Tous les temps correspondent à la somme des temps *user* et *system* en faisant une moyenne sur plusieurs exécutions. Les étoiles (* ou **) indiquent qu'un test n'a pas été effectué.

	CAML			Standard ML		Scheme
DS 3100	CAML V2.6-1	ZINC	CeML	SML/NJ	SML2C	SchemeToC
Fib	6,7	42	**2,5**	4,7	14,5	10,5
Tak	18,5	12,4	**0,7**	4,6	11,3	2,4
Iblt	7,5	41	**3,4**	9,1	25,2	8,5
Ibgt	32	170	**3,4**	8,8	24,5	19
Makestring	4,1	**0,3**	**0,3**	1,9	2,7	*
CountStr	12,5	**1,3**	1,9	6,6	10,3	*
TakR	*	14,7	**2,2**	5,5	12,4	8
Rev	14,6	9,6	**2,2**	2,6	6,4	2,3
SigmaMap	1	10,7	**0,6**	1,1	2,1	2,1
ItList	4,6	7,2	3,1	**2,1**	4	4,4
Double	5,4	10,4	6,4	**1,2**	4,8	7,1
TakE	24,2	18,3	15,4	**7,2**	14,5	*
SigmaVect	**	27 (9,8)	**1,1 (0,6)**	**	**	**
SigmaVectW	4,6	29 (11,7)	**1,3 (0,8)**	5,1	9,9	*

FIG. 1.1 – Tests simples de CeML sur MIPS

Les résultats sur machine SUN3 (SPARC) sont moins bons d'un facteur entre 2 et 4 ce qui correspond à peu près à la différence de puissance de ces deux machines. Les tests SPECint et SPECfloat [144] à l'époque indiquaient ce rapport de performances.

	CAML			Standard ML		Scheme
SUN 3/280	CAML V2.6-1	ZINC	CeML	SML/NJ	SML2C	SchemeToC
Fib	10,8	84	**10**	12,9		39,1
Tak	31,3	54	**3,3**	10,9		7,8
Iblt	**12,4**	153	12,8	22,4		35,3
Ibgt	56,4	501	**12,8**	22,5		115,6
Makestring	6,2	0,8	**0,4**	6,4		*
CountStr	20,5	**3,7**	4,7	31,7		*
TakR	*	82,1	**15**	*Bus Error*		53,8
Rev	24,5	39,1	**4,3**	4,4		7,1
SigmaMap	2,27	45,6	**1,4**	2,6		7,3
ItList	7,9	26,3	10,5	**3,7**		11,6
Double	9,5	45,8	23	**3,4**		21,7
TakE	40	83,9	44	**16,5**		*
SigmaVect	**	117 (41)	**4,2 (2,8)**	**		**
SigmaVectW	7,8	125 (49,7)	**4,4 (3,1)**	14,7		*

FIG. 1.2 – Tests simples de CeML sur SPARC

La deuxième série de tests correspond à des programmes plus conséquents ce qui permet de mesurer les implantations en condition «réelle». La figure 1.3 provient aussi de [30]

Les résultats des tests avancés (fig 1.3) montrent de bonnes performances pour les programmes fonctionnels et impératifs, mais celles-ci baissent quand les programmes deviennent très fonctionnels ou utilisent fortement

	CAML			Standard ML	
DS 3100	CAML V2.6-1	ZINC	CeML	SML/NJ	SML2C
Integr	4	6,7	**1,4**	**1,4**	3,8
Sieve	7,5	13,2	**2,4**	4	10,7
DivEuclid	29,5	25,4	17,5	**3,8**	9,8
KB	17,8	11,6	12	**2,6**	7,1
Soli_let	25,4 [1]	151(68)	**7,4 (4,6)**	29	*Erreur*

FIG. 1.3 – Tests avancés de CeML sur MIPS

des exceptions. Le mécanisme de `setjmp`/`longjmp` était assez coûteux.

On ajoute à cette série de tests le programme `Pseudoknot-96` qui a été utilisé pour tester une vingtaine de compilateur de langages fonctionnels [81].

	CAML		Standard ML
	Caml-Light	CEML	SML/NJ
Nucleic	59,4	9,3	8,1

FIG. 1.4 – Test nucleic sur SPARC

La représentation des flottants a une certaine importance sur ces résultats selon qu'ils prennent un mot (CeML) ou deux mots mémoire. La taille du programme ML original fait environ trois mille lignes, principalement des déclarations de données. Le C engendré par CeML est environ dix fois plus volumineux ce qui entraîne un temps de compilation en mode optimisé extrêmement long (plus d'une heure de l'époque).

Réactualisation des tests de performances

Si l'on exécute les tests précédents sur une machine récente on obtient des temps trop faibles pour être suffisant fiables dans leur appréciation. Pour cela les appels aux fonctions principales ont été modifiées pour obtenir des temps mesurables.

Comme les machines SUN3 ou DECstation ne sont plus disponibles, il est délicat de comparer les anciens tests de performance avec les nouveaux. Néanmoins comme le compilateur Caml-Light (anciennement nommé ZINC) existe sur les nouveaux processeurs, il nous est possible de comprendre les gains de performance par rapport aux résultats de Caml-Light.

La figure 1.5 montre les résultats obtenus pour Pentium/Linux. Les temps ont été obtenus par la moyenne de plusieurs essais mesurés par la commande `time`.

	Caml-Light	Objective Caml bc	Objective Caml natif	CeML
Integr	1,9	1,4	0,8	0,3
Sieve	4,1	2,6	0,8	1,2
DivEuclid	16,3	13	1	9,4
KB	8,2	6,6	0,8	4,7
Soli_let	35,4	10	0,5	1,2

FIG. 1.5 – Nouveaux tests de CeML sur Pentium

Sur les machines Pentium on pouvait s'attendre à un facteur de gain proportionnel à la fréquence utilisée pour l'ensemble des compilateurs. C'était sans compter sur les améliorations apportées aux compilateurs C, tout particulièrement `gcc`.

Les résultats de CeML sont toujours entre la version byte-code et la version native du compilateur Objective Caml. Quand les programmes deviennent très fonctionnels ou dans le cas d'utilisation massive d'exception les performances de CeML se rapprochent de la version byte-code. Dans les autres cas les performances sont plus proches de la version native pour Pentium. Comme la version byte-code d'Objective Caml est toujours meilleure que la version de Caml-Light, on obtient l'ordre suivant :

$$Caml - Light \leq ObjectiveCaml - bc \leq CeML \leq ObjectiveCaml - pentium$$

	Caml-Light	Objective Caml bc	Objective Caml nat	CeML
Nucleic (Pentium)	5,9	5,2	0,55	0,9

FIG. 1.6 – Nucleic sur Pentium

Remarque Il n'était pas question ici de lancer une discussion sur les performances de ces différents compilateurs CAML, mais d'essayer d'évaluer les gains de performances obtenus sur une longue période.

Les tests réactualisés vont servir au compilateur O'CAMiL qui produit des exécutables pour la nouvelle plate-forme d'exécution .NET.

1.3 Compilation vers .NET : O'CAMiL

Le compilateur O'CAMiL [108] est un compilateur compatible Objective Caml pour la plate-forme .NET [140]. Son but est de faciliter la diffusion d'applications Objective Caml et de comprendre les possibilités d'interopérabilité entre différents langages compilant vers .NET, en particulier C# [140]. Ce premier propotype de compilation d'Objective Caml vers .NET ne cherche pas l'optimisation à tout prix, mais plutôt la compatibilité avec le langage Objective Caml à la différence de FSHARP.

La plate-forme .NET[3] se veut un creuset où s'intègrent différents langages dans un cadre commun tant du point de vue du système de types CTS (*Common Type System*) que de l'environnement d'exécution CLR (*Common Language Runtime*). Chaque compilateur de langage produit un code intermédiaire portable MSIL (*MicroSoft Intermediate Language*) qui est ensuite assemblé et exécuté. Une application .NET ne s'intéresse pas aux langages sources des composants qu'elle manipule. Leur conformité au système de types garantit leur interopérabilité.

La liste des langages portés sur cette plate-forme est conséquente. On peut citer de grands classiques comme C#, J# ou A#[4] mais aussi des langages moins connus comme Eiffel[5], Scheme[6], Standard ML[7], F#[8], Mercury[9].

Bien que l'implantation principale .NET fonctionne uniquement sous Windows, des initiatives «Open Source» voient le jour comme Rotor[10] pour Windows et Unix BSD, et Mono[11] pour Linux. On rejoint alors le slogan de Java «COMPILE ONCE, RUN EVERYWHERE».

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

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

Pour apporter notre pierre à l'édifice et pour vérifier les propriétés annoncées on se propose de réaliser un compilateur .NET pour notre langage de prédilection, Objective Caml[100].

L'amélioration des techniques de compilation des langages fonctionnels sur les vingt dernières années permet de produire du code efficace pour des processeurs réels ou des machines virtuelles spécialisées. Ces techniques ne sont pas toujours adaptées aux nouvelles machines virtuelles pour langages objets comme la JVM (*Java Virtual Machine*) ou le CLR de .NET. En effet pour garantir la sûreté du chargement dynamique de code-octet, l'environnement d'exécution de ces machines effectue des vérifications de typage sur le code à charger. L'adage «Runtime Tags Aren't Necessary» de [8] n'est plus de mise.

Le compilateur O'CAMiL[12] se veut un vecteur de diffusion du langage Objective Caml. Sa principale contrainte est alors la compatibilité avec Objective Caml. Pour cela il dérive une nouvelle branche de production de code au sein même du compilateur Objective Caml. Il rencontre alors les difficultés de génération du code IL typé dans la mesure où Objective Caml, une fois un programme typé, n'utilise plus cette information dans le code produit. Il devient donc nécessaire de reconstruire les informations de type au niveau d'un des langages intermédiaires d'Objective Caml, sachant que le programme d'origine était typé. D'autres difficultés, plus classiques, apparaissent pour la compilation de la partie fonctionnelle du langage vers une machine virtuelle spécialisée pour les langages objets. Les portages de la bibliothèque standard (`stdlib`) et de la bibliothèque `graphics` ont montré l'interopérabilité avec le langage IL et les bibliothèques existantes. Comme le compilateur O'CAMiL est principalement écrit en Objective Caml, un test important de compatibilité a été réalisé par son auto-amorçage (*bootstrap*). Ainsi O'CAMiL peut être distribué sous forme d'une application .NET.

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

1.3.1 Plate-forme .NET

La plate-forme .NET est présentée comme la prochaine technologie de référence dans le développement d'applications Windows, autant pour les applications de bureautique que de serveurs Web ou encore toutes sortes de composants logiciels. Elle est censée apporter de nouvelles garanties en matière de gestion de sécurité et d'erreurs, et permettrait par exemple de remédier aux travers de Windows en matière de bibliothèques partagées[140] [13].

On s'intéresse ici uniquement à cette plate-forme à travers l'œil du développeur, et plus spécifiquement du développeur d'un compilateur. La plate-forme propose un environnement d'exécution CLR essentiellement composée d'une machine virtuelle à pile[14] et d'une bibliothèque de support BCL[15]. La machine virtuelle vérifie que du code-octet MSIL respecte le modèle de classes typées CTS. Ce code-octet est ensuite expansé vers du code exécutable par l'environnement d'exécution comme présenté à la figure 1.7. Un paquetage PE (*Portable Executable*) contient des instructions MSIL avec d'éventuels fichiers de ressources.

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

[12] `www.pps.jussieu.fr/~montela/ocamil`
[13] «The End of DLL Hell», `msdn.microsoft.com/netframework/`
[14] VES : Virtual Execution System.
[15] Base Class Library.

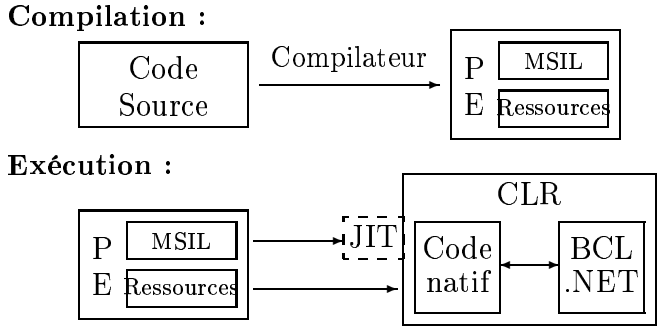


FIG. 1.7 – Compilation et exécution

Le code-octet MSIL

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

Déploiement

Le composant de base en .NET est l'assemblage (*a*ssembly). Un assemblage peut être signé à l'aide d'une clé cryptographique (*strong naming*) : la machine hôte peut alors accorder sa confiance au code qu'il contient, ce qui permet de partager ce code, en plaçant l'assemblage dans le cache global des assemblages[16], un répertoire spécial d'installation. Ce mécanisme est également exploité afin de faciliter la gestion de versions et de régionalisations distinctes de composants logiciels.

Sûreté à l'exécution

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

On note de plus que l'environnement est muni d'un mécanisme de récupération automatique de mémoire (*GC* de type *M*ark and Compact[140], ce qui affranchit le développeur des soucis de gestion de la mémoire.

[16]GAC : *Global Assembly Cache.*

Performances

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

Réflexion

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

1.3.2 Compilateur O'CAMiL

Schéma général

Notre objectif principal est de réaliser un portage de Objective Caml vers la plate-forme .NET totalement compatible avec l'implantation de référence, les soucis d'efficacité étant jugés secondaires dans un premier temps.
Adapter intégralement un langage fonctionnel évolué comme Objective Caml, possédant des traits impératifs et objets, avec un système de modules paramétrés et muni d'un système de typage statique avec inférence de types n'est pas chose facile. Notre expérimentation consiste à écrire un compilateur qui se branche de manière assez tardive sur le mécanisme de compilation standard de CAML, après les phases d'analyse grammaticale et d'inférence de types. Nous récupérons donc le programme à compiler déjà traduit vers un langage intermédiaire interne `Clambda` du compilateur CAML traditionnel, comme illustré sur la figure 1.8. `Clambda` explicite la gestion de l'environnement des fermetures. Nous introduisons une nouvelle représentation intermédiaire typée `Tlambda` : sa nécessité est discutée à la sous-section 1.3.2.

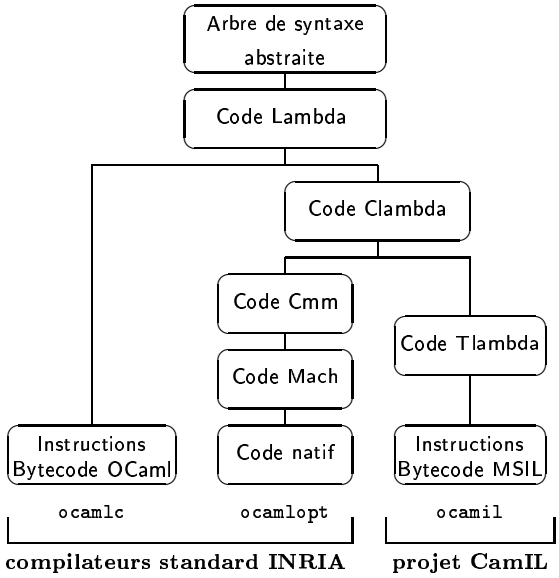


FIG. 1.8 – Branchement de O'CAMiL sur le compilateur Objective Caml

O'CAMiL transforme un ensemble de modules CAML en un fichier exécutable portable, comme sur la figure 1.7. Le code produit délègue un certain nombre de tâches de bas-niveau (gestion de mémoire, entrées-sorties

...) à l'environnement CLR et ses bibliothèques de support.

Représentation des données

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

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

Valeurs structurées Les n-uplets, les tableaux, les enregistrements, les listes et les valeurs de types sommes, sont traditionnellement représentés en mémoire par des blocs dans le tas, caractérisés par leur taille en mots et un octet de *t*ag (qui sert par exemple à coder le constructeur d'une valeur d'un type somme).

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

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

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

Contrôle de l'exécution

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

Exceptions Les exceptions CAML sont directement implantées en tirant parti du mécanisme d'exception de la plate-forme cible, au moyen d'une classe `CamIL.Exception` qui hérite de `System.Exception` (la classe mère de toutes les exceptions dans le CTS).

(Re)-typage de langage intermédiaire

Le compilateur O'CAMiL récupère un programme CAML sous forme de code `Clambda`. Ce code n'a plus d'information de type, et pire encore, il utilise certaines optimisations qui exploitent les caractéristiques de l'environnement d'exécution standard. Par exemple l'implantation CAML traite de manière homogène les valeurs entières et les pointeurs sur des blocs en les distinguant par un *bit* de *tag*. Ainsi l'allocation d'un bloc CAML dont une des cases contient un entier ne nécessite pas d'indirection sur cet entier, qui peut être mis en ligne dans la structure du bloc. En revanche de par la représentation des blocs comme tableaux d'objets

en MSIL, il est nécessaire d'allouer une encapsulation de l'entier.
Ce genre de manipulation complique la génération de code, et réclame une analyse du code Clambda source, visant à reconstituer une information partielle sur les types mis en jeu. Le tableau suivant montre une génération de code MSIL incorrecte parce qu'elle est ignorante des types (la variable `t` désigne un tableau) :

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

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

Interface extérieure

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

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

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

La version O'CAMiL consiste en un soupçon de code CAML autour de la fonction `GetEnvironmentVariable` de .NET.

1.3.3 Premiers résultats

Distribution O'CAMiL 0.1

La première version du compilateur O'CAMiL est disponible[12] dans sa version Windows. Nous avons adapté une grande partie de la bibliothèque standard de Objective Caml ainsi que la bibliothèque `graphics`.

Les traits fonctionnels, impératifs et objets[17] du langage ont été implantés, ainsi que le système de modules (foncteurs, compilation séparée).

[17] Voir la page WEB[12] de Camil pour le détail de l'implantation.

Performances

Il y a une vraie difficulté à mesurer des temps d'exécutions sous les versions récentes du système Windows. On ne retrouve pas de commande permettant de mesurer les temps passés dans le processus "user" et dans les appels systèmes (à la manière de la commande `time` d'Unix). L'utilisation de `time` sous `cygwin`[18], une sur-couche Unix sous Windows, n'est pas satisfaisante dans la mesure où le temps calculé ne concerne que le *thread* principal. De plus le mécanisme de JIT introduit une différence notable entre la première exécution d'un programme et les suivantes. Pour ces raisons nous mesurons le temps «réel» passé dans l'exécution d'un programme .NET. Ce temps est une moyenne de trois exécutions au delà de la première.

Il est intéressant de comparer ces mesures par rapport au compilateur de code-octet d'Objective Caml qui nous sert de référence. Deux autres compilateurs vers .NET sont utilisés : F# qui est compatible avec le noyau impératif/fonctionnel d'Objective Caml et SML.NET qui compile le noyau de SML[107].

On peut grouper les tests suivants en deux groupes :

- les petits tests comme `Integrate` (calcul flottant), `Sieve` (listes d'entiers) et `DivEuclid` (calcul de termes).
- les tests correspondant à des applications comme `Boyer` (calcul de termes, appels de fonction), `KB` (calcul de termes, fonctionnel et exceptions) et `Nucleic` (calcul flottant et arbres). Ce dernier test sert de référence dans [81] pour tester une vingtaine de compilateurs de langages fonctionnels.

Les tests suivants ont été réalisés sur un Pentium IV 2,4GHz sous Windows XP. Ils ont été construits de manière à obtenir des temps proches d'une seconde pour le compilateur natif (`ocamlopt`). On cherche à

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

FIG. 1.9 – Tests de performance (temps réel en secondes).

comparer les résultats de O'CAMiL avec le compilateur de code-octet (`ocamlc`). En dehors des exemples très fonctionnels les temps de O'CAMiL sont proches de ceux d'`ocamlc` (facteur $1,5$). Par contre on obtient de moins bons résultats pour un programme très fonctionnel comme `KB` (facteur 25).

Il est à noter que les compilateurs F# et SML.NET sont encore moins bons dans ce cas là que O'CAMiL. Cela provient des valeurs fonctionnelles et non pas du polymorphisme car ces deux compilateurs simplifient les contraintes de polymorphisme. F# compile vers une extension de MSIL, appelée ILX [146], introduisant la généricité. SML.NET à la manière de MLj [18] effectue au moment de l'édition de liens une analyse sur l'ensemble du programme pour le rendre monomorphe. Cette dernière optimisation, qui donne de bons résultats pour les programmes peu fonctionnels, empêche la compilation séparée.

La comparaison O'CAMiL et F# donne environ un facteur 2 à l'avantage de ce dernier. Le typage du code intermédiaire `Tlambda` de O'CAMiL n'est pas assez précis. Cela entraîne une allocation de structures de données plus importante que prévue (tableau d'objets) ce qui non seulement en ralentit l'accès mais nécessite des opérations de vérification de type dynamique coûteuses. Pour éviter cela une information supplémentaire provenant de la phase de typage du programme source devient nécessaire quand on compile vers une machine typée [98].

[18] `www.cygwin.org`

1.3.4 Travaux connexes

L'idée de pouvoir compiler des parties de programme issues de plusieurs langages vers la même plate-forme n'est pas nouvelle, y compris dans le monde fonctionnel typé. Plusieurs compilateurs vers C de ML ou de Scheme ont vu le jour au début des années 90 [32], [147], [137] facilitant l'interopérabilité à travers C. Néanmoins choisir comme cible une machine abstraite liée à un environnement d'exécution gérant la récupération mémoire et les ruptures de contrôle (exceptions) offre un bien meilleur niveau de sécurité à l'exécution, tout particulièrement dans le cadre d'asssemblage de code provenant de différentes sources. La JVM et le JRE (*Java Runtime Environment*) associé assurent ce niveau de sécurité. C'est d'ailleurs une des raisons du succès de la plate-forme Java. Avant la plate-forme .NET on pouvait déjà utiliser des langages fonctionnels typés statiquement (comme MLj [18]) ou dynamiquement (comme Bigloo [137]) compilés vers la JVM. Le code produit respecte les conventions de typage du code-octet de la JVM en facilitant l'interopérabilité de ces langages avec Java.

Cette idée de faciliter l'interopération entre langages n'est pas récente. Les langages de définition d'interface (*IDL*) tant pour CORBA[19] que pour COM[20] sont légions, y compris pour les langages fonctionnels comme HDIRECT[60] pour Haskell[21] ou OCAMLIDL[22] pour Objective Caml. La plate-forme .NET est plus récente que celle de Java, son assembleur portable IL est plus riche du point de vue typage (Types-Valeurs) et contrôle (appels de méthodes via une indirection), tout en simplifiant le mécanisme de JIT. Dans les deux cas se posent des difficultés pour compiler des traits de programmation différents du modèle Java, C# comme les fermetures (ML, Scheme, Haskell), l'héritage multiple (Eiffel, Objective Caml) et les continuations Scheme.

1.3.5 Travaux futurs sur O'CAMiL

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

Du point de vue typage il est regrettable de perdre les informations inférées par le compilateur. Tout d'abord l'efficacité en souffre en encapsulant/désencapsulant inutilement les valeurs manipulées. Ensuite cette imprécision nuit à la qualité de mise au point comme l'inspection de valeurs en cours d'exécution mais aussi pour la précision d'un mécanisme de persistance. Le retypage du code intermédiaire d'Objective Caml s'apparente à la décompilation de code-octet [6] où seules certaines informations de typage sont nécessaires. L'idée est alors d'ajouter dans le code intermédiaire d'Objective Caml, principalement au niveau des fermetures, une information de type superficielle permettant une reconstitution précise.

L'utilisation de bibliothèques C#, ou de bibliothèques Objective Caml à partir de C#, doit être réalisée sans modifier la nature du langage fonctionnel/impératif/objet qu'est Objective Caml. Pour cela, nous étudions actuellement un *IDL* simplifié, comme présenté au chapitre 3, et extensible pour l'intersection des modèles objet d'Objective Caml et de C#.

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

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

1.4 Conclusion : types à la compilation

Les deux compilateurs CeML et O'CAMiL utilisent des techniques de compilation très différentes. Dans le premier cas, CeML possède une information de types précise qui est propagée par un constructeur de types fonctionnels qui contient l'arité attendue de la fermeture. Dans le deuxième cas, O'CAMiL se branche au niveau du langage intermédiaire `clambda` du compilateur Objective Caml. L'information du typage est en grande partie perdue et ne peut pas être utilisée pour engendrer un code relativement efficace. C'est pour cela que le langage `clambda` est ensuite transformé en `tlambda` suite à un retypage sommaire de `ulambda`. De plus les langages cibles ne sont pas de même nature du point de vue des types. C permet de définir un type générique α (`char *`) tel que toute valeur immédiate (`int`) ou pointeur peut être contrainte vers ce type et inversement. Le langage intermédiaire MSIL de .NET différencie les valeurs immédiates des instances de classe. Le type α peut être codé par la classe `Object` mais cela oblige d'encapsuler les valeurs immédiates quand cela est nécessaire dans des instances de sous-classes d'`Object` comme pour les entiers `BoxInt`. On retombe sur les éternels problèmes de *boxing/unboxing* [96] que le typage du langage `ulambda` parvient en partie à régler. Néanmoins les structures de valeurs immédiates (listes d'entiers, tableaux d'entiers, ...) ou les environnements des fermetures sont plus gros qu'habituellement et l'accès est plus long d'où des performances grandement améliorables.

On en conclut que l'information de types issue de la phase de typage de la compilation est nécessaire pour produire un code efficace pour certaines machines abstraites comme le langage intermédiaire MSIL de .NET ou pour un langage cible de haut niveau comme le C engendré par CeML. D'autres essais l'ont montré comme le compilateur expérimental Gallium [97]. Il peut sembler dommage que l'actuel typeur d'Objective Caml ne s'en serve pas, mais il faut rappeler que le compilateur natif d'Objective Caml est orienté machine réelle (ce qui autorise les transtypages nécessaires). Néanmoins qui peut le plus peut le moins, et la conservation de l'information de typage devrait à terme être conservée dans les langages intermédiaires successifs pour faciliter la production de code vers les machines abstraites récentes. Le prix à payer est peut être trop important dans la mesure où la complexité du typeur d'Objective Caml va en croissant à chaque nouvelle version :

- 2.04 : stabilisation de l'extension objet
- 3.00 : labels, variants [69], ...
- 3.06 : méthodes polymorphes
- 3.07 : modules mutuellement récursifs, types privés

mais le fait de conserver cette information de typage dans les différentes phases de production de code peut aussi forcer la réécriture du typeur Objective Caml devenu un peu trop complexe par les ajouts récents.

Dans d'autres cas l'information de types calculée à la compilation peut être nécessaire. C'est le cas pour la reconstruction dynamique de types comme présenté page 87 qui nécessite de connaître l'information statique de types issue de la compilation pour reconstruire dynamiquement le type des valeurs tout particulière dans le cadre de fonctions polymorphes. Seule la combinaison de l'information statique de types avec la structure mémoire d'une valeur (ou bien la conservation d'une partie de l'histoire du calcul) permet de déterminer le type d'une valeur. Ce qui autorise son exploration tant pour les outils de mise au point que pour garantir un mécanisme de persistance sûr.

Chapitre 2

Extensions et compilation vers ML

On s'intéresse dans ce chapitre aux possibilités d'extension du langage Objective Caml et aux outils syntaxiques disponibles, en particulier `camlp4` [54],[55]. Une des principales difficultés dans la construction d'extensions d'un langage est de les faire évoluer en même temps qu'évolue le compilateur du-dit langage. En effet la modification des sources d'un compilateur nécessite de fabriquer un *patch* à chaque nouvelle distribution. Or si l'on modifie trop profondément le compilateur original, chaque nouveau *patch* demande un effort important que les auteurs ne peuvent pas toujours fournir. Dans la mouvance d'Objective Caml on peut citer deux exemples :

- Olabel [68] qui ajoutait les labels et a suivi les nouvelles distributions d'Objective Caml de la version 2.04 à 3.04, puis a été intégré au langage dès les versions 3.05.
- Jocaml [45] ajoutant le modèle du *join-calculus* au langage Objective Caml dont la dernière implantation date d'avril 2003, la précédente de mai 2000, toutes les deux sont basées sur le compilateur Objective Caml 1.07 (de décembre 1997).

Il semble difficile de tenir le rythme des nouvelles versions (environ deux par an) sans intégration à la distribution.

Pour cela l'idée est de compiler les extensions vers un programme Objective Caml, quitte à adjoindre de nouvelles bibliothèques en ML ou en C pour les tâches de bas niveau. Une extension produira ainsi du code ML compilable si les nouvelles versions restent compatibles ascendantes. L'outil `camlp4` [54], intégré depuis la version 3.03 aux distributions du langage fournit le cadre pour de telles extensions.

On le montre par la suite en décrivant un compilateur du noyau du langage Scol utilisé dans le projet EDICA pour expérimenter des extensions à Scol. On compile un programme Scol en construisant un programme Objective Caml équivalent. On ajoute ensuite au noyau Scol un mécanisme d'exceptions, les déclarations de fonctions locales et de types paramétrés. Ce travail a été réalisé en liaison avec Anne-Gwenn Bosser et Francisco Alberti de l'équipe de R&D de Cryo-networks.

On applique ce schéma d'extension pour le langage Objective Caml lui-même. Pour tenter de structurer les hiérarchies de classes on introduit la relation *instanceof* entre les objets et les classes. Pour étendre le modèle objet d'Objective Caml on autorise une relation de *downcast* vérifiant les relations d'héritage et de sous-typage. Cette vérification dynamique de types est utilisée pour implanter, avec l'aide d'Eric Delpech de St Guilhem, une persistance objet sûre.

Cette technique est ensuite reprise pour tenter de simplifier une extension data-parallèle, appelée Caml-Flight réalisée en 1995 par Christian Foisy et l'auteur. Cette implantation nécessitait de modifier profondément la machine d'exécution de Caml-Light ainsi que le compilateur. Les nouveaux traits de programmation introduits dans Objective Caml : les processus légers, la persistance et les communication réseau, alliés à `camlp4` simplifient et rendent portable cette implantation. Celle-ci reprend et étend une première version réalisée par Bruno Baia et Clément Garrigou sous ma direction.

2.1 scocaml

Le langage Scol [86],[23] a été conçu par la société Cryo-networks en 1997 pour la conception d'univers persistants sur Internet (autant dans le domaine des jeux vidéo en lignes, des sites de E-Commerce à immersion 3D et des communautés virtuelles). Pour pouvoir utiliser cette technologie dans tous ces domaines, elle a pris la forme d'un langage de programmation applicatif typé de communications muni de nombreuses API multimédia. C'est une vision opposée à celle des navigateurs VRML qui visualisent des scènes 3D et ajoutent ensuite du multi-utilisateurs comme Communauty Place [142].

C'est à notre connaissance un des rares exemples de l'utilisation du modèle fonctionnel typé pour des applications grand public commercialisées et vantant ce type de technologie, d'où l'intérêt qu'il a suscité. De ces contacts a été conçu le projet Edica (Environnement de Développement Interactif (grand public) pour la Conception d'Agents dans des mondes virtuels) qui a été labellisé RNTL[1] en avril 2001. Son but est d'accroître les possibilités de développement d'univers virtuels. Pour cela la technologie Scol doit être enrichie, tout d'abord au niveau du langage lui-même qui doit être le plus réutilisable et fournir des outils de programmation fiables et puissants. Au-delà du langage, la technologie SCOL se doit de permettre la conception d'agents artificiels autonomes pour « peupler » les univers virtuels développés. Le projet est donc décomposé en deux sous parties : l'évolution propre du langage et la définition d'un environnement de développement pour la conception des agents artificiels.

Outre la définition d'une nouvelle génération de la technologie SCOL, des agents artificiels pourront être aisément définis par les concepteurs de sites SCOL, rendant les univers virtuels plus vivants. Ceci est valable quelque soit la vocation du site, ludique, commerciale ou communautaire, et permettra d'étendre la viabilité de ces univers.

Pour une bonne séparation des tâches des différentes équipes du projet Edica, il a été décidé de construire un prototype du noyau du langage Scol principalement pour faciliter la réalisation des extensions au langage Scol prévues dans le projet.

Ce prototype se veut une plateforme expérimentale pour l'intégration de nouveaux paradigmes de programmation au langage Scol. Son indépendance vis-à-vis du produit d'origine de Cryo-networks autorise à mener les tâches en parallèle dans les différentes équipes.

On présente tout d'abord les caractéristiques du noyau applicatif du langage Scol pour apprécier la construction du protoype `scocaml`. Celui-ci est construit en utilisant `camlp4` ce qui facilitera les extensions tant du point de vue du typage que du contrôle.

2.1.1 Description du langage Scol

Le langage Scol est un langage impératif et applicatif[2] dans la mesure où bien qu'il puisse manipuler des valeurs fonctionnelles comme valeurs du langage, il ne possède pas de déclarations locales de fonctions et de plus il a besoin d'une forme spéciale pour l'appel de valeurs fonctionnelles. C'est un langage statiquement typé, polymorphe paramétrique avec inférence de types à évaluation immédiate. Le polymorphisme paramétrique n'apparaît que pour les déclaration globales de fonctions et non pas pour la déclaration de types. Les définitions de type englobent les types produits (enregistrements) et les types sommes (à la ML). Son système de types intègre les types récursifs au sens de [52]. La partie impérative autorise les modifications physiques de variables, des tableaux et des n-uplets. Toutes les variables (globales, locales, paramètres, motifs) peuvent être modifiées. En dehors de l'appel de fonction les principales structures de contrôle sont la séquence, la conditionnelle, le filtrage de motifs et les itérations. Il ne possède pas de mécanisme d'exceptions. Pour gérer un calcul non abouti, Scol définit une valeur spéciale `nil`.

Au niveau de la communication, Scol définit des constructeurs de communication permettant de passer des arguments locaux à une fonction distante. Le résultat de ce calcul n'est pas utilisé.

[1] Réseau National des Technologies Logicielles.
[2] forme dégradée des langages fonctionnels.

Un programme Scol est une suite de déclarations globales.

Au final Scol subit des influences de ML, Lisp et Id[111].

Un programme Scol est chargé dynamiquement, compilé en code-octet puis ce code-octet est interprété. La machine virtuelle est une machine à pile. Outre l'interprète de code-octet, son environnement d'exécution possède un gestionnaire de mémoire (GC), et une bibliothèque d'exécution contenant la gestion des communications.

Son implantation est maintenant «Open Source»[3]. Sa communauté de développeurs a constitué une association [4] pour promouvoir cette technologie.

Comme le langage est «petit», on présente sa grammaire des déclarations et des expressions sous forme BNF[5] pour ensuite détailler la sémantique du typage. Les figures 2.1, 2.2, 2.3 décrivent respectivement la syntaxe des types, des déclarations globales et des principales expressions du langage Scol tel qu'elle est décrite dans [86].

Scol distingue les déclarations de types contenant une variable de type `un` (`Type`) des types monomorphes (`Typemono`) qui n'en contiennent pas.

$$
\begin{array}{|l|}
\hline
Type \ ::= B \mid \mathbf{u_n} \mid \mathbf{r_n} \\
\quad \mid tabType \mid [\ Type* \ ] \mid \textbf{fun} [\ Type* \ ]\ Type \\
\hline
TypeMono \ ::= B \mid \mathbf{r_n} \\
\quad \mid tabTypeMono \mid [\ TypeMono* \ ] \mid \textbf{fun} [\ TypeMono* \ ]\ TypeMono \\
\hline
\end{array}
$$

avec B = Type de base, u_n = variable liée et r_n = récursion de niveau n

FIG. 2.1 – Déclarations de types

Les types «produits» sont représentés par des n-uplets : `[ I F ]` correspond au type `int * float` d'Objective Caml. Les niveaux de récursion indiquent un cycle dans le graphe d'un type pour l'expression d'un type «cyclique». Cela permet de nommer une partie d'une déclaration de types sans avoir à déclarer un autre type. Par exemple les listes peuvent être représentées par un couple dont le premier élément est un entier et le deuxième un tel couple. La liste vide est représentée par la valeur `nil`. Un type pour ces listes est `[ I r1 ]`[6].

Les déclarations globales de Scol comprennent les déclaration de types (`typeof`, `struct` et `typedef`), de fonctions (`fun`), de variables (`var`), de constructeurs de communication (`defcom` et `defcomvar`) et les déclarations avancées (`proto`).

$$
\begin{array}{|rl|}
\hline
Scol & ::= Definition* \\
Definition & ::= \textbf{fun}\ Function\ (\ Args\ )\ =\ Program\ ;; \\
& \mid \textbf{typeof}\ Var\ =\ TypeMono\ ;; \\
& \mid \textbf{var}\ Var\ =\ Val\ ;; \\
& \mid \textbf{struct}\ NewType\ =\ [\ Fields\ ]\ Function\ ;; \\
& \mid \textbf{typedef}\ NewType\ =\ TypeConstr\ ;; \\
& \mid \textbf{defcom}\ Com\ =\ string\ \{\mathbf{I},\mathbf{S}\}\ *;; \\
& \mid \textbf{defcomvar}\ Comvar\ =\ \{\mathbf{I},\mathbf{S}\}\ *;; \\
& \mid \textbf{proto}\ Function\ =\ Type\ ;; \\
\hline
\end{array}
$$

FIG. 2.2 – Déclarations globale

[3] `www.sourceforge.net`

[4] `www.scol-technologies.com`

[5] ...

[6] équivalent au type `(int * 'a) as 'a` d'Objective Caml.

Scol appelle `Program` le corps d'une fonction. Outre les expressions arithmétiques et logiques, les accesseurs/modificateurs des enregistrements et des vecteurs, on rencontre les expressions suivantes : application d'une fonction, déclaration locale, boucle, modification physique d'un n-uplet, application d'une valeur fonctionnelle et filtrage de motifs.

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{Expr} \mid \mathit{Expr}\ ;\ \mathit{Program} \\
\mathit{Expr} & ::= & \mathit{Arithm} \mid \mathit{Arithm}\ ::\ \mathit{Expr} \\
\mathit{Arithm} & ::= & A_1 \mid A_1\ \&\&\ \mathit{Arithm} \mid A_1\ ||\ \mathit{Arithm} \\
A_1 & ::= & A_2 \mid !\ A_1 \\
\ldots & & \\
A_6 & ::= & \mathit{Term} \mid -\ A_6 \mid\ \ A_6 \\
\mathit{Term} & ::= & (\ \mathit{Program}\ ) \mid (\ \mathit{Program}\ )\ ; \\
 & & \mid \{\ \mathit{Program}\ \} \mid \{\ \mathit{Program}\ \}\ ; \\
 & & \mid \mathit{int} \mid \text{`}\mathit{char} \mid \mathbf{nil} \\
 & & \mid \mathit{string} \mid [\ \mathit{Arithm}*\ ] \\
 & & \\
 & & \mid \mathit{Var}(.\mathit{Term})* \mid \mathbf{set}\ \mathit{Var}(.\mathit{Term})* = \mathit{Term} \\
 & & \mid \mathit{Var}(.\mathit{NameOfField})* \mid \mathbf{set}\ \mathit{Var}(.\mathit{NameOfField})* = \mathit{Term} \\
 & & \\
 & & \mid \mathit{Function}\ \mathit{ArgsFunction} \mid @\ \mathit{Function} \\
 & & \\
 & & \mid \mathbf{let}\ \mathit{Arithm} -> \mathit{Locals}\ \mathbf{in}\ \mathit{Arithm} \\
 & & \mid \mathbf{if}\ \mathit{Arithm}\ \mathbf{then}\ \mathit{Arithm}\ \mathbf{else}\ \mathit{Arithm} \\
 & & \mid \mathbf{while}\ \mathit{Arithm}\ \mathbf{do}\ \mathit{Arithm} \\
 & & \mid \mathbf{mutate}\ \mathit{Arithm} <-\ [\ \{,\mathit{Arithm}\}*\ ] \\
 & & \mid \mathbf{exec}\ \mathit{Arithm}\ \mathbf{with}\ \mathit{Arithm} \\
 & & \\
 & & \mid \mathit{Constr}\ \mathit{Arithm} \mid \mathit{Constr}0 \mid \mathbf{match}\ \mathit{Arithm}\ \mathbf{with}\ \mathit{Case}
\end{array}
$$

FIG. 2.3 – Expressions

La règle d'application des fonctions masque une grammaire dynamique dans la mesure où l'application attend le nombre d'arguments indiqués dans sa déclaration.

Principales différences entre Scol et Objective Caml

La transformation d'un programme Scol en un programme Objective Caml équivalent se heurte à de réelles difficultés dans la mesure où les deux langages diffèrent sur de nombreux points. Les particularités les plus importantes du langage sont les suivantes :

- typage :

  bien que les deux langages soient typés statiquement avec inférence de types et polymorphes paramétriques, le langage Scol utilise systématiquement les types cycliques (en déclarant le niveau du cycle) pour la définition de structures de données dynamiques.

- variables :

  toutes les variables Scol sont physiquement modifiables à tout moment de l'exécution

- valeur à tout faire `nil` :

  la valeur `nil` de Scol est utilisée comme valeur par défaut pour tous les types ; elle peut être retournée par toute fonction dont le calcul n'a pas abouti et être utilisée comme argument optionnel.

– en Scol les n-uplets peuvent aussi être physiquement modifiés
– Scol autorise les déclarations avancées (*proto*) de fonctions
– chargement dynamique et compilation à la volée
– les bibliothèques initiales diffèrent fortement

Ces principales différences montrent que la traduction pose des difficultés tant au niveau du typage que des valeurs physiquement modifiables.

2.1.2 Prototype `scocaml`

Le but de ce prototype est de pouvoir expérimenter des extensions au langage de base. L'outil `camlp4` est dans ce cas fort pratique par son mécanisme d'extension de grammaire. La grammaire initiale de `scocaml` est donc pensée dans ce but.

Organisation

Les sources du prototype Scocaml se découpent en deux parties :

1. compilateur
   – `env_scol` : environnement de compilation
   – `pa_scol.ml` : analyseur lexical et grammaire Scol
2. bibliothèque d'exécution
   – `mutate.c` : pour les opérations particulières de bas niveau
   – `std_scol.ml` : bibliothèque d'exécution Scol

L'ensemble de ce traducteur est constitué de moins de trois mille lignes Objective Caml auxquelles il faut ajouter une centaine de lignes de C.

Au final on obtient un *toplevel* pour le noyau de Scol :

toplevel scocaml

```
> ./scocaml

        scocaml : Scol en O'Caml, projet Edica, 0.2b
        using
        Objective Caml version 3.06

        Camlp4 Parsing version 3.06

# var e = 1;;
val e : int ref = {contents = 1}
# e + 1;;
- : int = 2
# fun add(a,b) = a + b;;
val add : int -> int -> int = <fun>
#  add 2 3;;
- : int = 5
# exec @add with [ 2 3 ];;
- : int = 5
# #quit;;
```

Principaux choix d'implantation

On cherche à construire un *toplevel* acceptant en entrée des phrases Scol ; celles-ci sont traduites en Objective Caml puis compilées et exécutées par Objective Caml. Le *toplevel* est augmenté de la bibliothèque d'exécution Scol.

lexer La grammaire telle qu'elle est définie aux figure 2.2 et 2.3 distingue l'arité des fonctions déclarées dès la phase d'analyse lexicale. Pour cela on ajoute dans les unités lexicales les fonctions d'arité de 0 à 9. Dès sa déclaration le nom d'une fonction correspondra à une unité lexicale dépendante de son arité.

grammaire On ajoute comme points d'entrée principaux outre les `definition` Scol la possibilité d'évaluer une expression (program) ou une directive au sens Objective Caml.

extrait de la grammaire scol

```
program:
  [[ e = expr  -> e
   | e = expr; ";"; p = program -> <:expr< do  ignore($e$); $p$ >>
  ]]
;

expr:
  [ RIGHTA
   [ a = arithm -> a
   | a = arithm; "::"; e = expr -> <:expr<(Std_scol.cons $a$ $e$)>>
   ]
  ]
;

arithm:
  [ NONA [ a = a1 -> a
   | a = a1; "&&"; b = arithm -> <:expr<(Std_scol.op_and $a$ $b$)>>
   | a = a1; "||"; b = arithm -> <:expr<(Std_scol.op_or $a$ $b$)b>>
  ]]
;
```

types cycliques Les niveaux de cycle dans les déclarations de type Scol sont traduits vers des types cycliques Objective Caml. Par exemple le type des listes Scol vues comme des couples : `[u0 r1]` se traduit :

types cycliques

```
> ocaml -rectypes
        Objective Caml version 3.06

#  type 'a nlist = ('a * ('a nlist));;
type 'a nlist = 'a * 'a nlist
# let cons a b   = ((a,b) : 'a nlist);;
val cons : 'a -> 'a nlist -> 'a nlist = <fun>
```

effets de bord Toutes les variables globales sont des références. Les variables locales deviennent aussi des références si elles sont modifiées dans la portée de leur déclaration, sinon elles conservent leur type. Dans le cas des paramètres de fonctions, si ceux-ci sont modifiés dans le corps de la fonction, des déclarations locales correspondant à des références du même nom sont ajoutées au début du corps de la fonction en masquant ainsi les paramètres initiaux. Si un tel paramètre est transmis dans un appel à une autre fonction, seule la valeur référencée est alors passée. On reste dans un cadre classique d'appel par valeur.

déclarations avancées Les fonctions déclarées par `proto` sont traduites par des références sur une fonction du même type déclenchant systématiquement une exception. Toute utilisation d'une telle fonction fait un accès à cette référence.

nil Pour chaque type τ prédéfini ou déclaré, on ajoute à l'ensemble des valeurs du type τ la valeur nil_τ. Ainsi `1 + nil` est une expression bien typée qui retourne `nil`. On implante tous les `nil` possibles en lui donnant le type $\forall\alpha.\alpha$. Cela implique que toutes les fonctions doivent tenir compte de cette possible valeur. Par exemple la fonction qui retourne la tête d'une liste peut s'écrire de la manière suivante :

nil
```
let nil = ref 0;;

let hd (nl : 'a nlist) =
  let nil_nlist = Obj.magic nil in
    if nl == nil_nlist then Obj.magic nil
    else match nl with (x,_) -> x;;
```

affichage De par la valeur `nil` et les types cycliques, l'afficheur d'Objective Caml ne peut pas être utilisés dans ces cas. On modifie l'afficheur par une directive `install_printer` en utilisant une des fonctions prédéfinies :

printers
```
external print_scol : 'a -> int -> int -> 'b -> unit = "print";;
let print_val x =
  print_scol nil 1  1 x ;;

let no_print_val x = ();;
```

2.1.3 Extensions

Ces extensions portent sur les trois caractéristiques suivantes :
- le contrôle d'exécution : par un mécanisme de traitement d'exceptions
- la manipulation de valeurs fonctionnelles : par la définition de fonctions locales ; celles-ci peuvent être retournées comme résultat d'une fonction. Dans ce cas il y a construction d'une véritable valeur fonctionnelle.
- le polymorphisme paramétrique : par la définition de types paramétrés par le programmeur.

Le codage de ces extensions est bien séparé du code du prototype initial.

On présente ces trois extensions ainsi qu'un exemple complet les mettant en œuvre.

Exceptions

Il y a trois nouvelles constructions syntaxiques pour les exceptions qui s'inspirent fortement d'Objective Caml :
- la déclaration d'exceptions, qui peuvent être vues comme un type somme extensible monomorphe :

  syntaxe : `exception` constructeur `;;`

- la pose d'un récupérateur d'exceptions :

  syntaxe : `try` expression `with` filtrage

- le déclenchement d'une exception :

  syntaxe : `raise` expression

On retrouve les contraintes de type classiques :
- l'expression passée à `raise` doit être une exception
- le type de l'expression protégée par le `try` est le même que les expressions que l'on retrouve dans les branches du filtrage associé.

Le concision du code correspondant à cette extension montre la puissance des outils utilisés :

exception

```
EXTEND

  definition: LEVEL "top"
    [[ "exception"; e = typeconstr' ->
         let (c,pc) = e in
         make_constr_sum loc [e];

         <:str_item<exception $c$ of $list:pc$>>
    ]]
  ;
  arithm:
    [[ "try"; a = arithm; "with"; c = case ->
        <:expr< try $a$ with [ $list:c$] >>
     | "raise"; a = arithm -> <:expr< raise $a$>>
   ]]
   ;
END;;
```

L'implantation est facilitée dans la mesure où le mécanisme d'exception existait en Objective Caml et que le filtrage simplifié de Scol avait déjà été implanté dans le prototype initial.

On construit alors une boucle d'interaction intégrant cette extension. Voici le déroulement de celle-ci sur un petit exemple :

session

```
> ./scocamlext

        scocaml : Scol en O'Caml, projet Edica, 0.2b
        using

        Extended version 0.2b-A

        Objective Caml version 3.06

        Camlp4 Parsing version 3.06

#  exception Div_by_zero;;
exception Div_by_zero
# fun div_aux (a,b) =
    if ( b == 0) then raise Div_by_zero else a / b;;
val div_aux : int -> int -> int = <fun>
# fun div (a,b) =
    try div_aux a b
    with (Div_by_zero -> 0);;
val div : int -> int -> int = <fun>
# div 8 2;;
- : int = 4
# div_aux 8 0;;
Exception: Div_by_zero.
```

Les exceptions seront utilisées dans les exemples de constructions de structures de données polymorphes des extensions suivantes.

Fonctions locales et fermetures

La documentation du langage Scol le présente comme un langage fonctionnel, c'est-à-dire comme un langage manipulant des valeurs fonctionnelles (fermetures). Une valeur fonctionnelle peut être représentée comme un couple contenant le pointeur du code à exécuter et l'environnement dans lequel ce code s'exécute. En

Scol, comme il n'y a que des fonctions globales, cet environnement est toujours vide. Il existe néanmoins un mécanisme autorisant l'application partielle d'une fonction à moins d'arguments qu'attendus, mais il n'offre pas la pleine fonctionnalité de tels langages.

Pour cela, une deuxième extension a été implantée, ajoutant au langage la définition de fonctions locales. Une telle fonction peut être retournée comme résultat de l'application de sa fonction englobante. Dans ce cas, une valeur fonctionnelle est construite avec un environnement potentiellement non-vide.

Une fonction locale n'a de portée que dans l'expression qui la suit :

syntaxe : `fun` nom (param_1, ..., param_n) = expression_1 `in` expression_2

Le source du programme `map.pkg` de la figure 2.4 montre l'utilisation d'une fonction locale dans la définition de `map` dans un style fonctionnel classique.

fichier map.pkg

```
fun map(f,l) =
  fun map_aux (l2) =
    if (l2 == nil) then nil
    else  ((exec f with
        [(hd l2)]) :: (map_aux (tl l2)) )
  in
    map_aux l;;

fun f(a,b) =
  fun g(x) =  x + a
  in
    g a;;

fun succ (x) = x + 1;;
```

suite

```
typeof l = [I r1];;
typeof r = [I r1];;

fun make_l() =
  set l = (1::2::3::nil);;

fun doit() =
  make_l;
  map @_fooI l;
  set r = map @succ l;
  map @_fooI r;;
```

FIG. 2.4 – Déclaration locale de fonctions

Les déclarations locales n'apparaissent pas dans les informations de typage des déclarations globales. Voici le typage inféré par le prototype quand le fichier `map.pkg` est chargé. La fonction locale `map_aux` n'est pas visible à l'extérieur de la fonction `map` même si la valeur fonctionnelle correspondante est autorisée à effectuer une extension de portée.

typage de map.pkg

```
value map : ('a -> 'b) -> Std_scol.nlist 'a -> Std_scol.nlist 'b =  <fun>
value succ : int -> int = <fun>
value l : ref ((int * 'a) as 'a) = <fun>
value r : ref ((int * 'a) as 'a) = <fun>
value make_l : 'a -> unit = <fun>
value doit : 'a -> Std_scol.nlist unit
```

L'exécution suivante correspond à l'appel de la fonction `doit`. Ce programme affiche la liste `l` contenant trois éléments, puis applique la fonction `map` sur `succ` et `l`, le résultat est alors rangé dans la liste `r` qui est ensuite affichée.

exécution

```
# doit;;
3
2
1
4
3
2
- : Std_scol.nlist unit =
```

La fonction `map` appelle sa fonction locale `map_aux` qui contient dans son environnement de calcul le paramètre fonctionnel `f` qui vaut `succ` dans cet exemple.

Là encore l'extension est assez simple dans la mesure où le langage cible (Objective Caml) accepte ce type de définition. Pour ce faire, on étend la règle `arith` de la grammaire Scol, correspondant aux expressions, comme définie dans [86].

Types paramétrés utilisateurs

Le langage Scol intègre une inférence de types qui calcule le type le plus général d'une expression ou d'une déclaration. Par exemple le type de la fonction `identite` qui prend un argument et le retourne tel quel possède en Scol le type suivant : `fun [u0] u0` indiquant que le type du paramètre est quelconque, et le type du résultat est du même type que celui du paramètre. Cette généralisation des types fonctionne pour les fonctions mais il n'est pas possible au programmeur de définir de nouveaux types paramétrés. L'extension réalisée cherche à rendre Scol plus générique.

L'exemple de la figure 2.5 définit la structure de pile homogène où tous les éléments de la pile sont du même type : La figure 2.5 définit une structure contenant un champ couple `c` correspondant au sommet de pile et

fichier stack.pkg

```
struct t<u0> = [ c : [u0 r1] ] mkt;;
exception Empty;;

fun create() = mkt nil;;
fun clear (s) = set s.c = nil;;
fun copy (s) = mkt s.c ;;
fun push (x,s) = set s.c = (x :: s.c);;
fun pop (s) =
  if s.c == nil then raise Empty
  else
     let hd s.c -> r in (set s.c = tl s.c; r);;
```

FIG. 2.5 – Définition d'un type paramétré

à la suite de la pile.

La syntaxe choisie est la suivante :

syntaxe : `type` nom <parametredetype> = declarationdetype

Dans l'exemple de la figure 2.5 le paramètre de type est `u0`, nommage déjà employé dans l'inférence de types pour les fonctions Scol.

La définition de types paramétrés accepte les déclarations de type produit et de type somme comme dans l'exemple de la figure 2.6 page 41.

2.1.4 Exemple complet

L'exemple de la figure 2.6, page 41, montre l'utilisation des différentes extensions dans un même programme. Ce programme définit une structure de queue homogène utilisant deux types paramétrés : un type somme pour les éléments de la queue (`queue_cell`) et un type produit (`t`) pour conserver le début et la fin de la queue. L'exception `Empty` est définie pour gérer la prise d'un élément dans une queue vide. La fonction `iter` contient une fonction locale `iter_aux` qui effectue l'itération sur la structure.

fichier queue.pkg

```
exception Empty;;

typedef  queue_cell<u0>  = Nil
                  | Cons [ u0 queue_cell<u0> ];;

struct t<u0> = [ head : queue_cell<u0>,
                 tail : queue_cell<u0> ] mkt;;

fun create() = mkt [Nil Nil];;

fun clear (q) = set q.head = Nil;
  set q.tail = Nil;;

fun add(x,q) =
  match q.tail with
    (Nil -> let Cons [x Nil] -> c in
       (set q.head = c; set q.tail = c ))
  | (Cons y -> let Cons [x Nil] -> c in
       (mutate y <- [ _ c ]; set q.tail = c)) ;;

fun peek (q) =
  match  q.head with
     (Nil -> raise Empty)
  |  (Cons [ x _ ] -> x) ;;
```

suite

```
fun take(q) =
  match q.head with
    (Nil -> raise Empty)
  | (Cons [x rest] -> (set q.head = rest;
       (match rest with
          (Nil -> set q.tail = Nil)
        | ( _ -> nil));
      x)) ;;

fun length_aux (q) =
  match q with
    (Nil -> 0)
  | (Cons [ _ rest] -> 1 + (length_aux rest)) ;;

fun length (q) = length_aux q.head;;

fun iter(f,q) =
  fun iter_aux(q) =
    match q with
      ( Nil -> nil)
    | ( Cons [ x rest] ->
         (exec f with [x]; iter_aux  rest))
  in
    iter_aux q.head;;
```

FIG. 2.6 – Définition de types paramétrés : produit et somme

Ce programme est traduit vers le langage cible Objective Caml qui infèrera le type des déclarations globale comme suit :

typage de queue.pkg

```
exception Empty
type queue_cell 'a = [ Nil | Cons of ('a * queue_cell 'a) ]
type t 'a =  head : mutable queue_cell 'a; tail : mutable queue_cell 'a
value mkt : (queue_cell 'a * queue_cell 'a) -> t 'a = <fun>
value create : 'a -> t 'b = <fun>
value clear : t 'a -> unit = <fun>
value add : 'a -> t 'a -> unit = <fun>
value peek : t 'a -> 'a = <fun>
value take : t 'a -> 'a = <fun>
value length_aux : queue_cell 'a -> int = <fun>
value length : t 'a -> int = <fun>
value iter : ('a -> 'b) -> t 'a -> 'c = <fun>
```

Sur cet exemple on s'aperçoit des nouvelles possibilités de programmation offertes par ces extensions. On peut citer un meilleur contrôle des situations particulières par le mécanisme d'exceptions, la puissance et la

concision de la programmation fonctionnelle par la manipulation des fermetures, et une plus grande généricité introduite par les types paramétrés.

2.2 Extension objet : `coca-ml`

Objective Caml possède une extension objet [128] qui apporte un langage de classes au noyau fonctionnel/impératif de Caml. Une déclaration de classe construit un nouveau type «objet» et une fonction de construction d'instances de ce type. Les classes peuvent être en relation d'héritage et en relation de sous-typage. Ces deux relations sont bien distinctes. 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.

Il faut considérer le langage Objective Caml comme un langage muni d'une extension objet et non pas comme un langage à objets. Néanmoins grâce à la liaison tardive il permet de structurer des composants logiciels plus réutilisables [24]. Malheureusement cette extension ne permet pas de construire facilement des hiérarchies de classes générales. En particulier certains modèles de conception classiques [67] comme les fabriques ou les modèles à délégation à la AWT de Java ne peuvent pas être définis. Dans ces cas il n'est pas toujours possible de connaître le type exact retourné par l'appel d'une méthode : seul le type d'une classe ancêtre est connu comme type du résultat. Pour résoudre ce problème il est nécessaire d'ouvrir le dogme du typage statique en autorisant une vérification dynamique de type en cas de contraintes de types (*downcast*) entre objets.

Cette nouvelle extension, appelée `coca-ml`[7] (pour «Cast objet en caml») réalisée grâce à l'outil `camlp4` [54] autorise le «cast» dynamique d'objets d'une classe `c` vers une autre classe `d` si d'une part le type de la classe `d` est bien sous-type du type de la classe `c` et d'autre part si `d` est bien une sous-classe, au sens de l'héritage, de la classe `c`. La première propriété est vérifiée statiquement au typage, la deuxième propriété nécessite une vérification dynamique à l'exécution. Il devient donc possible après avoir contraint le type d'un objet de classe `d` vers le sur-type de la classe `c`, de pouvoir de nouveau considérer cet objet comme de classe `d` ou de toute classe intermédiaire entre `c` et `d` de l'arbre d'héritage. Pour être sûr qu'une classe `d` est bien sous-type de toutes les classes dérivées de `c` vers `d` il est nécessaire de le vérifier soit au moment de l'affaiblissement de types («upcast»), soit au moment de la définition de la classe `d` en testant que cette classe est bien sous-type des classes dont elle hérite. Ces deux techniques ont été implantées mais cette dernière a notre préférence car elle ne modifie pas l'opérateur d'affaiblissement (`:>`) du langage tout en étant plus facile à utiliser et en étant plus efficace.

2.2.1 Héritage et sous-typage en Objective Caml

La définition d'une classe Objective Caml définit une abréviation d'un type objet et une fonction de construction d'instances de ce type. Un type objet correspond à une suite de noms de méthodes associés à leur type :

$$< m_1 : \tau_1; m_2; \tau_2; ...; m_n : \tau_n >$$

Les types τ_i peuvent être fonctionnels ou objets, et les types objets peuvent être récursifs et paramétrés. Les variables d'instance n'apparaissent pas dans le type engendré par la définition d'une classe.
Les deux relations principales entre les classes sont :

- la relation structurelle d'héritage ;
- la relation de sous-typage entre les types des classes.

Bien que certains langages objets confondent ces deux relations en une seule, Objective Caml les distingue.

[7] `www.pps.jussieu.fr/~emmanuel/Public/Dev/coca-ml`

Relation d'héritage

Lorsqu'une classe `d` est définie en héritant de la classe `c`, toutes les méthodes apparaissant dans l'abréviation de type `c` sont présentes dans l'abréviation de type d. Si `d` définit de nouveaux noms de méthodes, celles-ci apparaîtront dans le type inféré. Si `d` redéfinit une méthode héritée, celle-ci devra avoir le même type que la méthode héritée : il n'y a pas de surcharge en Objective Caml. Néanmoins une méthode redéfinie peut contenir le type en cours de définition dit «type de `self`».
Supposons que le type suivant a été inféré par la déclaration de la classe `c` :

$$c =< m1 : \alpha \rightarrow \tau_1; m2 : \tau_2 \rightarrow \alpha > \ as\ \alpha$$

On note une définition récursive du type. Ce n'est pas un type paramétré car la variable α n'apparaît pas dans le terme gauche et est liée par le `as`.

Si maintenant on déclare une classe `d` qui hérite de `c` et qui définit une nouvelle méthode `m3` de type int, on obtient le type suivant :

$$d =< m1 : \alpha \rightarrow \tau_1; m2 : \tau_2 \rightarrow \alpha; m3 : int > \ as\ \alpha$$

La classe `d` a hérité des méthodes définies dans `c`, mais comme ces méthodes contenaient le type de l'instance (type de `self`) dans leurs types celui-ci correspond au type `d` dans la définition de `d`. Le fait d'hériter des méthodes contenant le type de `self` change en fait le type de ces méthodes dans la mesure où le type de `self` change.

Sous-typage

La relation de sous-typage est une relation réflexive et transitive entre types.

Un type objet d $= < m1 : \tau'_1; ...; m_p : \tau'_p; m_{p+1} : \tau'_{p+1}; ...; m_n : \tau'_n >$ est un sous-type d'un type objet c $= < m1 : \tau_1; ...; m_p : \tau_p >$ si tous les noms de méthodes apparaissant dans c existent aussi dans d et tous les types des méthodes m_i (pour i allant de 1 à p) de d sont sous-type des types de ces mêmes méthodes dans c. Cette relation est notée $\leq$.
Comme le type d'une méthode est quelconque cette relation s'applique à tous les types du langage de types. Les types de base ne sont sous-type que d'eux-mêmes. La relation vérifie pour les constructeurs `*` (paire) et `->` (fonctionnel) les propriétés suivantes :

- si $\tau_i \leq \tau'_i$ alors $(\tau_1 * \tau_2) \leq (\tau'_1 * \tau'_2)$
- si $(\tau'_1 \leq \tau_1)$ et $(\tau_2 \leq \tau'_2)$ alors $(\tau_1 \rightarrow \tau_2) \leq (\tau'_1 \rightarrow \tau'_2)$

Le constructeur flèche est co-variant à droite et contra-variant à gauche.
Soit un objet o de type τ_1, si $\tau_1 \leq \tau_2$ alors il est possible d'utiliser l'objet o en lieu et place de valeurs de type τ_2. Cette contrainte de type est explicite dans le langage Objective Caml et s'exprime de la manière suivante : `(expr : tau1 :> tau2)` indiquant que si `expr` est du type τ_1 et si $\tau_1 \leq \tau_2$ alors `expr` est vu comme du type τ_2. Une fois l'objet `o` considéré de type τ_2, il est impossible de le reconsidérer comme un objet de type τ_1 si les types sont différents.

Cette contrainte de typage se retrouve dans la littérature francophone sous différentes appellations selon le point de vue adopté. Au niveau des types on parlera d'«affaiblissement» du type, au niveau des classes de «sur-classage». Si on intègre les notions d'héritage et de types on peut employer «transtypage ascendant» ou «upcast».

Héritage n'est pas sous-typage

Ces deux relations sont de natures différentes. En Objective Caml elles ne sont pas équivalentes, en particulier l'héritage ne garantit pas le sous-typage. Cela provient des méthodes où apparaît le type de `self` dans un type fonctionnel et de la contra-variance à gauche du constructeur fonctionnel flèche.

Si on construit une classe `c` dont le type engendré est $c =< eq : \ 'a \rightarrow bool > \ as \ 'a$ alors une classe `d` qui hérite de `c` et qui définit une nouvelle méthode `m` n'est pas sous-type de `c`. En effet pour que $d \leq c$ il faut que le type de la méthode `eq` de `d` soit sous-type du type de la méthode `eq` de `c`. C'est-à-dire $d \rightarrow bool \leq c \rightarrow bool$. De par la contra-variance à gauche cela implique que $c \leq d$. Ce qui n'est pas vrai car la méthode `m` de `d` n'existe pas dans `c`.

Le seul cas où l'héritage conserve la relation de sous-typage en présence de méthodes comportant le type de `self` en position paramètres est quand les types des classes sont identiques. Par contre quand les classes ne possèdent pas de telles méthodes, alors l'héritage donne bien aussi la relation de sous-typage.
C'est justement cette propriété qui va permettre de proposer une extension au langage Objective Caml pour faire du *downcast* sur les objets. On trouve là aussi plusieurs traductions comme les termes de «transtypage descendant», «sous-classage», «spécialisation» ou «rétrécissement» ou encore «renforcement».

2.2.2 Extension `coca-ml`

Le but principal de l'extension `coca-ml` est d'autoriser le *downcast* objet sans casser le système de types du langage. Cela implique de devoir effectuer une vérification de type à l'exécution. L'information de types d'un programme Objective Caml est perdue une fois le programme correctement typé. Cela nécessite de conserver un minimum d'informations pour connaître la classe de construction d'un objet, l'appartenance d'une classe à un arbre d'héritage de classes et la vérification de la relation de sous-typage entre classes.

On introduit alors les opérateurs suivants :
- `cc(o)` : donne la classe de construction de l'objet `o` ;
- `d` *hérite de* `c` : qui indique que `d` vaut `c` ou bien que `d` a `c` comme ancêtre dans l'arbre d'héritage ;
- `d` *sous-hérite de* `c` : équivalent à `d` *hérite de* `c` et $d \leq c$.

La relation de sous-héritage est elle aussi réflexive et transitive.

Pour ce faire chaque classe possède une clé unique de type *id* sous forme d'une déclaration globale et chaque déclaration de classe ajoute les cinq méthodes de la figure 2.7 dans le corps de la classe.

méthode	type	action
`get_id`	*id*	retourne la clé de la classe de construction
`check_id`	$id \rightarrow bool$	teste si la classe de construction de l'objet est identique à l'argument
`check_instanceof_id`	$id \rightarrow bool$	teste si la classe de construction de l'objet est une sous-classe de l'argument ;
`check_subinstanceof_id`	$id \rightarrow bool$	teste si la classe de construction est en relation de sous-héritage avec la classe passé en argument
`check_subclass`	$id \rightarrow id \rightarrow bool$	teste si la classe de construction d'un objet est en relation de sous-héritage avec le premier argument et que celui-ci sous-hérite du second.

FIG. 2.7 – Méthodes ajoutées

L'idée principale de `coca-ml` est d'autoriser une contrainte de type («cast») d'un objet `o` considéré comme du type c d'une classe `c` vers le type d d'une classe `d` si `cc(o)` *sous-hérite de* `d`. On distingue alors une contrainte de type ascendante (*upcast*) qui vérifie `cc(o)` *sous-hérite de* `c` et `c` *sous-hérite de* `d` d'une contrainte de type descendante (*downcast*) qui vérifie `cc(o)` *sous-hérite de* `d` et `d` *sous-hérite de* `c`. L'affaiblissement de type d'une contrainte ascendante se doit juste de vérifier la relation de sous-typage, ce qui est le cas par la relation de sous-héritage, y compris par affaiblissements successifs de par la transitivité de la relation de sous-héritage. Le renforcement de type d'une contrainte descendante considère l'objet du type de sa

classe d'origine puis l'affaiblit vers la classe cible. Il est à noter que du point de vue des types il n'est jamais dangereux de contraindre le type d'un objet vers le type de sa classe de construction. Cela garantit la correction de ces contraintes de types en particulier pour la contra-variance. La sémantique n'est pas forcément évidente pour le programmeur en présence d'héritage multiple et si l'on désire conserver l'emploi de l'opérateur d'affaiblissement actuel (`:>`). De plus ce mécanisme est totalement dynamique et ne satisfait pas forcément le programmeur Objective Caml qui apprécie la cadre du typage statique. Pour cela nous allons détailler la syntaxe et la sémantique de l'extension.

Syntaxe et sémantique

L'extension `coca-ml` ajoute au langage d'origine trois types de constructions :
- la relation de sous-héritage qui devient un nouveau champ des structures de classe ;
- des prédicats vérifiant des propriétés de la classe de construction d'un objet par rapport à une classe particulière ;
- des opérateurs de contraintes de type qui changent le type statique d'un objet sans modifier sa représentation.

Relation de sous-héritage La relation de sous-héritage (mot clé `subinherit`) s'emploie de la même manière que la relation d'héritage (`inherit`) dans le corps de la définition d'une classe : `subinherit c` $arg_1 \ldots arg_n$.
La classe en cours de définition hérite de la classe `c` et son type est sous-type de c.

Prédicats sur les relations On définit syntaxiquement trois prédicats qui vérifient que la classe de construction d'un objet est par rapport à la classe passée en paramètre respectivement «identique», «sous-classe» et en relation de «sous-héritage» comme présenté à la figure 2.8.

	`expr of c1`	`expr instanceof c1`	`expr subinstanceof c1`
Syntaxe	1/ `expr` est une expression 2/ `c1` est un identificateur de classe		
Typage	1/ type de `expr` : $< nom\_methode : id \rightarrow bool >$ avec `nom_methode`		
	`check_id`	`check_instanceof_id`	`check_subinstanceof_id`
	2/ le type de l'instruction est *bool*		
Evaluation	1/ `expr` vaut `o`		
	2/ appel de la méthode		
	`check_id`	`check_instance_id`	`check_subinstanceof_id`
	sur `o` avec comme argument la clé de la classe `c1`		

FIG. 2.8 – Prédicats sur les relations

Ces prédicats ne changent pas le type d'un objet mais permettent de vérifier qu'une contrainte de type pourra s'appliquer sur un objet.

Instructions de contrainte de type Les instructions de contrainte de type sur des objets ne modifient pas la valeur de l'objet sur lequel elles s'appliquent mais le forcent à être considéré d'un autre type. Elles sont présentées à la figure 2.9. En cas d'échec des vérifications des relations de classes à l'évaluation, une exception est déclenchée.
La première construction (`cast expr to d`) est la plus générale. Elle considère l'expression `expr` comme du type de la classe `d` et teste à l'exécution que la classe de construction de l'objet `o` résultat de l'évaluation de

	`cast expr to d`	`upcast expr from c to d`	`downcast expr from c to d`
Syntaxe	1/ `expr` est une expression		
	2/ `d` est un identificateur de classe		
	3/	`c` est un identificateur de classe	
Typage	1/	$expr : c$	
	2/	$c \leq d$	$d \leq c$
	3/ `expr` est alors considéré du type d		
Evaluation	1/ `expr` vaut `o`		
	2/ `cc(o)` sous-hérite `d`		
	3/	`cc(o)` sous-hérite de `c`	
	4/	`c` sous-hérite de `d`	`d` sous-hérite de `c`

FIG. 2.9 – Instructions de contraintes de type

`expr` est bien en relation de sous-héritage avec la classe `d`. On part du postulat qu'un objet peut toujours être contraint sans risque vers le type de sa classe de construction. En effet comme l'objet a déjà été considéré de ce type il peut l'être de nouveau. Au typage de vérifier que le contexte d'emploi l'autorise. Une fois l'objet considéré du type de sa classe d'origine, il est possible de contraindre son type vers toute classe `x` dont la classe de construction sous-hérite. La seule contrainte sur le type de `expr` est d'être un type objet possédant la méthode `check_subinstanceof` (voir figure 2.7).

On obtient sur l'exemple suivant un objet `o` de classe de construction `z`, qui est contraint une première fois vers la classe `a` puis ensuite vers la classe `m` :

cast
```
cast (cast (new z) to a) to m;;
```

La figure 2.10 montre les trois cas simples possibles si `z` est sous-classe de `a` et `m`. On présente la construction des classes sous la forme de schémas UML simplifiés dans la mesure où les variables d'instance n'apparaissent pas. On différencie la relation d'héritage habituelle (flèche simple) de la relation de sous-héritage (flèche barrée du symbole `<=`) dans les schémas de relation. Le premier cas correspond à deux «upcast» successifs ;

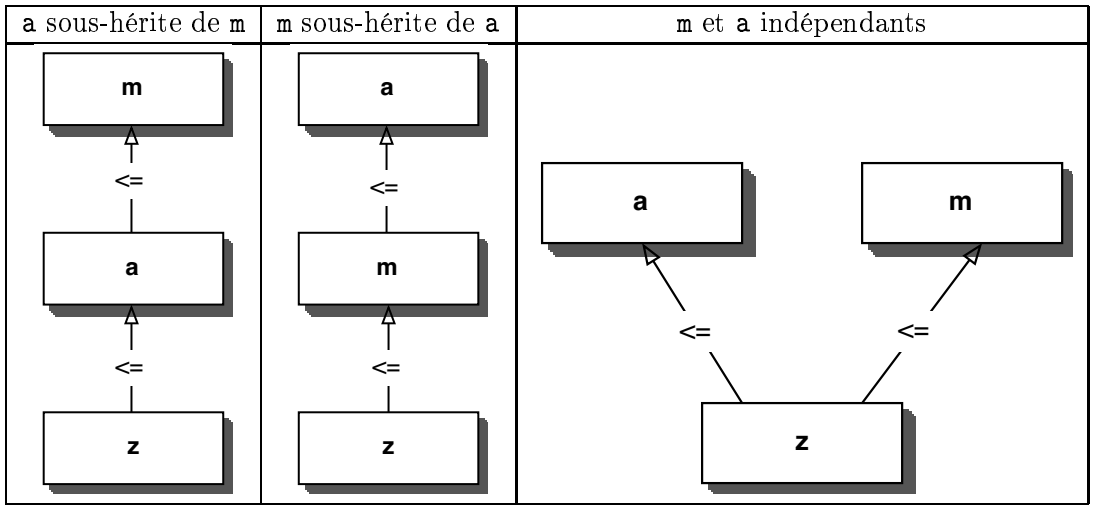


FIG. 2.10 – Trois graphes d'héritage possibles

le deuxième cas est le plus classique, c'est un «upcast» suivi d'un *downcast* vers une classe intermédiaire. Le

troisième cas est introduit par l'héritage multiple. Dans ces trois cas comme `z` sous-hérite de `a` et de `m` il n'y a pas de danger dans la contrainte.
Les deux autres constructions sont des restrictions de la première permettant plus de contrôle statique de types. Elles orientent le sens de la contrainte de type par rapport à l'arbre de sous-héritage. De plus le type de l'expression doit être compatible avec le type de la classe d'origine (`from`) de la contrainte. Enfin les classes d'origine et de destination (`to`) sont en relation de sous-typage cohérente par rapport au sens de la contrainte. Ces restrictions permettent de détecter au typage des contraintes qui ne pourront jamais aboutir. Si l'on avait écrit : `downcast (upcast (new z) from z to a) from a to m;;`, seul le deuxième cas de la figure 2.10 aurait été autorisé.

Compatibilité avec les opérateurs de contrainte et d'affaiblissement de types Le langage Objective Caml possède deux opérateurs de contrainte de types :

- l'opérateur « `:` » : `(expr : t)` qui unifie le type de `expr` avec le type `t`
- l'opérateur d'affaiblissement `:>` : `( expr :> d )`[8] qui considère `expr` de type `d` si $type(expr) \leq d$.

Ces deux opérateurs ne tiennent absolument pas compte des relations d'héritage entre classes. Ainsi un objet pourra être contraint ou affaibli vers le type d'une classe n'appartenant pas à sa hiérarchie de classes au sens de l'héritage.
Il est souhaitable de conserver la sûreté des nouvelles contraintes de types en présence d'objets contraints ou affaiblis par les opérateurs d'Objective Caml. Le cas général est celui d'un objet `o` de classe de construction `z` qui a été contraint vers le type de la classe `a` et qui va être contraint vers la classe `m` comme dans l'exemple suivant :

```
let o_z = new z;;
let o_a = ( o_z :> a);;
let o_m = cast o_a to m;;
```

La classe de construction de `o_z`, `o_a` et `o_m` est toujours `z` dans la mesure où un objet n'est pas modifié par une contrainte de type. On obtient les deux contraintes suivantes :

- $z \leq a$
- `z` *sous-hérite de* `m`

Les classes `a` et `m` ne sont pas forcément en relation de sous-typage dans la mesure où dans chacune d'elle peut apparaître une méthode n'existant pas dans l'autre. Néanmoins la contrainte de type reste sans danger si `z` *sous-hérite de* `m`. Certes la sémantique peut paraître un peu étrange dans le cas où la classe `a` n'est pas en relation d'héritage et de sous-typage avec `m`, mais ce qui compte est la relation de la classe de construction avec la classe de destination. On retrouve le cas 3 d'héritage multiple de la figure 2.10 pour la relation de sous-typage.

Exemples

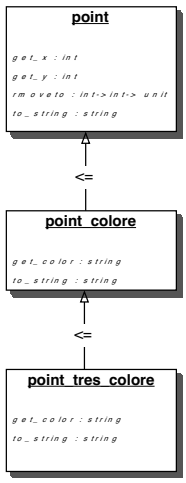
On présente trois petits exemples qui montrent les nouvelles possibilités de cette extension au langage. Le premier construit une hiérarchie linéaire à trois niveaux. Le deuxième montre comment traduire des méthodes binaires. Le dernier détaille un héritage multiple.

Plusieurs niveaux d'héritage La figure 2.11 reprend l'exemple inévitable sur l'héritage de points colorés. Il y a trois classes `point`, `point_colore` et `point_tres_colore` :

- `point_tres_colore` sous-hérite de `point_colore` qui sous-hérite elle-même de `point` ;

[8] On ne tient pas compte des différences sémantiques de la forme complète `(expr : c :> d)` dans la mesure où seule la contrainte de type vers `d` nous intéresse ici.

Hiérarchie



Programme

points

```
# let p1 = new point 2 3;;
val p1 : point = <obj>
# let p2 = new point 3 4;;
val p2 : point = <obj>
# let pc1 = new point_colore 1 2 "bleu";;
val pc1 : point_colore = <obj>
# let ptc1 = new point_tres_colore 4 5 "vert";;
val ptc1 : point_tres_colore = <obj>

# let p3 = upcast pc1 from point_colore to point;;
val p3 : point = <obj>
# let p5 = upcast ptc1 from point_tres_colore to point;;
val p5 : point = <obj>

# let pc3 = downcast p3 from point to point_colore;;
val pc3 : point_colore = <obj>
# let ptc3  = downcast p5 from point to point_tres_colore;;
val ptc3 : point_tres_colore = <obj>
# let pc6 = downcast p5  from point to point_colore;;
val pc6 : point_colore = <obj>
```

FIG. 2.11 – Exemple avec plusieurs niveaux d'héritage

On construit les instances `p1` et `p2` de la classe `point`, `pc1` de `point_colore` puis `ptc1` de `point_tres_colore`. `p3` (respectivement `p5`) est une instance de `point_colore` (respectivement `point_tres_colore`) considérée comme du type `point`.

On obtient donc trois *downcasts* possibles : `p3` est contraint en `point_colore`, `p5` est contraint en `point_tres_colore` et `p5` est contraint en `point_colore`.

Réécriture de méthodes binaires Pour conserver dans l'héritage la relation de sous-typage, il est nécessaire de masquer les méthodes possédant un type fonctionnel où le type de `self` apparaît en position paramètre. La figure 2.12 construit une classe `top` ayant une méthode `eq` qui teste l'égalité entre deux instances de `top` et deux sous-classes `left` et `right` qui hérite de `top` et redéfinissent la méthode `eq`.
La définition de la classe `right` diffère de la classe `left` par une variable d'instance `r` (à la place de `l`), une méthode `get_r` (à la place de `get_l`) qui retourne la valeur de `r` et surtout par la redéfinition de `eq` qui suppose recevoir une instance de `right` comme paramètre et effectue alors le «cast» vers `right` de l'argument.
Le petit programme suivant crée deux instances de `left` (`l` et `l2`) et une de `right` (`r`) puis effectue deux appels de la méthode `eq` sur `l` avec respectivement `l2` et `r` comme paramètre contraint au type `top`.

```
let l = new left 10;;
let l2 = new left 10;;
let r =  new right 10;;
if l#eq(cast l2  to  top) then print_string "OK"
else print_string "PB";;
try
 ignore(l#eq (cast r to  top));
  print_string "PB"
with Cast.Cast_failure _ -> print_string "OK";;
```

Ce style de programmation s'apparente plus au style Java avec ses avantages et ses inconvénients. Il est à noter que la méthode `clone` est du type α (type de `self`) et ne casse pas la relation de sous-typage de la hiérarchie de classe dans la mesure où la méthode n'a pas un type fonctionnel.

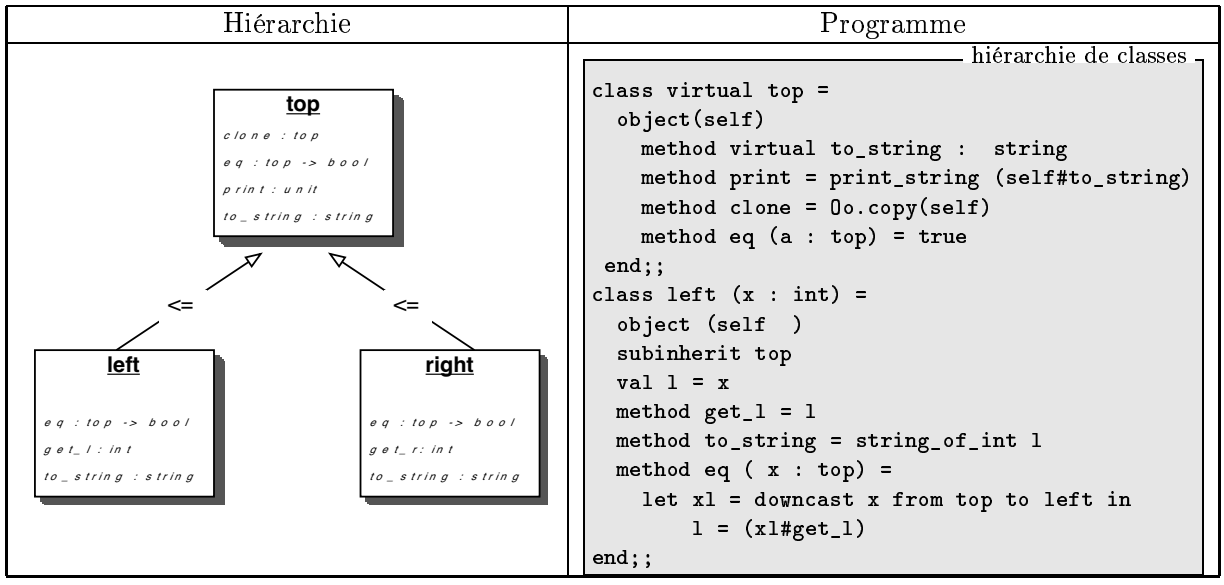


FIG. 2.12 – Exemple de premiers niveaux d'une hiérarchie

Héritage multiple Le troisième exemple (voir figure 2.13) montre un cas d'héritage multiple. La classe `abc` hérite des classes `a`, `b` et `c`. Ces trois dernières héritent de la classe `t`.

Le dernier calcul (`nr`) montre comment passer directement d'une instance de la classe `abc` vue comme du type `b` vers le type `a`. Cet exemple aurait pu s'écrire en deux étapes : `let nr = upcast (downcast b1 from b to abc) from abc to a;;`.

Implantation

Cette implantation ne touche pas le compilateur de la distribution d'Objective Caml. `coca-ml` transforme un programme Objective Caml en un autre programme Objective Caml. Pour cela elle utilise l'outil `camlp4`. Dans le cas présent la syntaxe du langage Objective Caml fournie avec `camlp4` est modifiée.

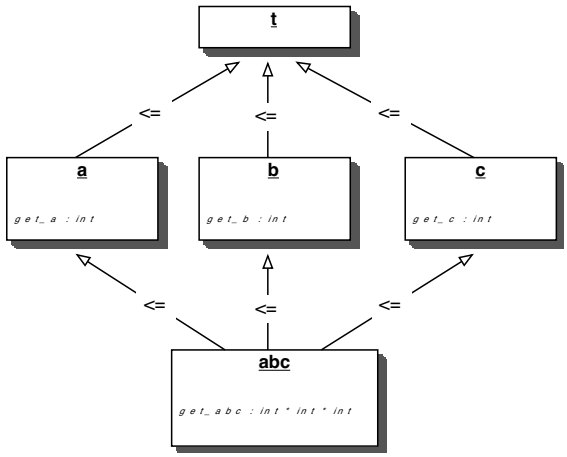
`coca-ml` modifie principalement la déclaration de classes et ajoute six nouvelles instructions qui manipulent les noms de classe (`of`, `instanceof`, `subinstanceof`, `cast`, `upcast` et `downcast`) aux expressions du langage ainsi que la déclaration de sous-héritage (`subinherit`) dans une déclaration de classe.

L'implantation a d'autre part un fichier `cast.ml` qui contient la définition du type `id` et de l'exception `Cast_failure` ainsi que les fonctions d'appel des méthodes de vérification sur un objet.

Déclaration de classe Une déclaration de classe `c` va maintenant créer une clé unique d'identification appelée `key_c`, de type abstrait *id*, et ajouter les méthodes publiques de la figure 2.7 à la classe.

Le choix d'implantation est de coder dans les objets les méthodes de vérification de propriétés d'héritage. Voici le code des méthodes principales ajoutées d'une classe `z` qui hérite d'une classe `a` et sous-hérite d'une classe `m`.

Hiérarchie



Programme

```
# let abc1 = new abc 1;;
val abc1 : abc = <obj>
# let a1 =  (upcast abc1 from abc to  a);;
val a1 : a = <obj>
# let b1 =  (upcast abc1 from abc to  b);;
val b1 : b = <obj>
# downcast b1 from b to abc;;
- : abc = <obj>
# let t1 = upcast a1 from a to t;;
val t1 : t = <obj>
# downcast t1 from t to a;;
- : a = <obj>
# let r = downcast t1 from t to abc;;
val r : abc = <obj>
# let nr = cast b1 to a;;
val nr : a = <obj>
```

FIG. 2.13 – Exemple d'héritage multiple

code produit

```
let ___z = (( Cast.make_id ()) : Cast.id );;
class z =
  object
    inherit a as ___super_0
    inherit m as ___super_1
    method ___check_instanceof_id (k : Cast.id) =
      (___z == k || ___super_1#___check_instanceof_id k)
                 || ___super_0#___check_instanceof_id k
    method ___check_subinstanceof_id (k : Cast.id) =
      (___z == k || ___super_1#___check_subinstanceof_id k)
    method ___check_subclass (k1 : Cast.id) (k2 : Cast.id) =
      if (___z == k1) then
        if (k1 == k2) then true else ___super_1#___check_subclass k2 k2
      else ___super_1#_check_subclass k1 k2
...
end;
fun ___x -> (___x : z :> m);;
```

Le préfixe `___` est ajouté à tout nouvel identificateur créé. Les noms des classes héritées sont conservés si ils existent ou engendrés dans le cas contraire.

Chaque classe possède une clé unique qui sera vérifiée physiquement dans les tests. En cas de sous-héritage une expression à l'extérieur de la classe vérifie la relation de sous-typage. Les classes simplement héritées apparaissent dans la méthode `check_instanceof_id` mais pas dans `check_subinstanceof_id`. Dans le cas où la classe n'hérite de personne, le corps de ces deux méthodes teste la clé d'identification de la classe qui a construit l'objet avec la clé passée en argument. Dans le cas où la classe hérite ou sous-hérite d'autres classes le corps de ces méthodes comprend les appels des méthodes de même nom des classes ancêtres sur l'objet lui-même avec la même clé comme argument. Comme la liaison des méthodes des classes ancêtres est statique, les tests s'effectueront bien avec la clé des classes ancêtres. La méthode `check_subclass` effectue une double remontée dans l'arbre de sous-héritage pour comparer l'ordre de la classe de construction avec les deux classes passées en paramètres.

Nouvelles instructions On distingue les instructions correspondant aux prédicats des relations des instructions effectuant effectivement une contrainte de types sur les objets.

- Les instructions `o of c1`, `o instanceof c1` et `o subinstanceof c1` sont traduites respectivement vers l'appel des méthodes `check_id`, `check_instanceof_id` et `check_subinstanceof_id` sur l'objet `o` avec la clé de `c1` passée en argument. En syntaxe simplifiée pour `o instanceof c1` cela donne : `o#check_instanceof_id(key_c1)`
- L'instruction `cast expr to c1` vérifie dynamiquement que la classe de construction de l'objet résultat de l'évaluation de `expr` est bien en relation de sous-héritage avec la classe `c1` :

magic
```
let o = expr in
  if o#check_subinstance_of(key_c1)
  then ((Obj.magic o ) : c1)
  else raise (Cast_failure loc ...)
```

- Les instructions `upcast expr from c2 to c1` et `downcast expr from c2 to c1` vérifient statiquement que `expr` a le type de `c2`, que la relation de sous-typage est bien satisfaite entre les deux classes `c1` et `c2`, et enfin que la relation de sous-héritage est bien vérifiée (dynamiquement) entre ces deux classes. Voici le code engendré pour `downcast expr from c2 to c1` :

code engendré
```
let o = ( expr : c1) in
  let x = ((Obj.magic o) : c1 ) in
    let _ = ( x :> c2) in
      if o#subclass key_c1 key_c2 then ( (Obj.magic o) : c2)
      else raise (Cast_failure ...)
```

Seules les instructions qui utilisent `magic` comportent un danger de transgresser le système de types. Le principal `magic` est celui qui considère l'objet du type de la classe de destination. Comme dans les trois instructions il y a la vérification que la classe de construction de l'objet est en relation de sous-héritage avec la classe destination, on garantit que l'objet peut être contraint vers ce type. Les autres instruction `magic` sont utilisées pour effectuer des restrictions statique en vérifiant la relation de sous-typage entre classe supposée d'origine et la classe destination.

Modules et classes Les méthodes supplémentaires sont ajoutées automatiquement aux signatures des modules. Seules les classes possédant une structure (`object .. end`) peuvent être traitées. La déclaration d'une classe en fonction d'une autre : `class b = a` ne possède pas assez d'informations pour être traités et pour différencier les clés de ces classes. La cohabitation d'objets classiques et d'objets étendus est possible tant que ces deux familles de classes ne sont pas en relation d'héritage et que les nouvelles instructions ne sont pas utilisées sur des objets classiques ce qui provoquerait une erreur de typage.

Les modules paramétrés peuvent être appliqués sur des structures possédant des déclarations de classes y compris dans le cas des modules locaux.

Limitations de l'extension La principale limitation de cette extension est de ne pas accepter des classes paramétrées dans les «cast». Mais ce problème ne peut pas être résolu dans le cas général. Prenons par exemple une classe `'a d` qui hérite d'une classe `c`. Si on crée une instance `o` de type *int d*, il est bien sûr possible de contraindre son type vers `c`. Si le type de cette instance est ensuite contraint vers la classe `d` quel est le type acceptable pour le paramètre de type du type `d` ? Vu la technique utilisée pour l'implantation actuelle rien n'interdirait d'écrire : `downcast o from c to float d` ce qui casserait la sûreté du typage.

Coût de l'extension Le choix de cette implantation a été de coder dans les définitions de classe les méthodes de vérification des héritages. La première conséquence est de faire grossir la taille du code par le code de ces méthodes supplémentaires. Il est à noter que la table des méthodes d'une classe est partagée par toutes les instances de cette classe. Les objets n'en seront pas plus gros pour autant. Seul le GC qui conserve la table des méthodes est légèrement ralenti dans la mesure où cette table occupe un peu plus d'espace. Il

n'y a pas de déclarations statiques de ces tables pour traiter de manière uniforme les classes créées par des modules locaux. Donc tant que l'on ne s'en sert pas le coût de l'extension est extrêmement faible.

Si l'on s'en sert, le coût est alors proportionnel à la profondeur de la hiérarchie de classes. Il est à noter que l'appel d'une méthode ne profite pas des optimisations de l'application totale de la partie fonctionnelle du langage ce qui est dommage dans la mesure où les méthodes supplémentaires sont toujours appliquées avec tous leurs arguments. Cela est déjà le cas pour les paramètres fonctionnels des fonctions et reste d'un coût accepté.

2.2.3 Persistance en `coca-ml`

La persistance est la conservation d'une valeur en dehors de l'exécution courante d'un programme. C'est le cas quand on stocke une valeur dans un fichier. Cette valeur est alors accessible par tout programme ayant accès à ce fichier. L'écriture et la lecture d'une valeur persistante nécessite la définition d'un format pour la représentation des données. En effet, on doit savoir passer d'une structure complexe en mémoire, comme un arbre binaire, vers une structure linéaire (une suite d'octets) rangée dans un fichier. C'est pourquoi le codage de valeurs persistantes s'appelle souvent «linéarisation».

L'implantation d'un mécanisme de linéarisation de structures de données doit effectuer certains choix et présente des difficultés que nous présentons ci-dessous.

- **lecture-écriture de structures de données.** Comme la mémoire peut toujours être vue comme un vecteur de mots, une valeur peut toujours correspondre à la mémoire qu'elle occupe, quitte à ne conserver que la partie utile en compactant alors la valeur.
- **partage ou copie.** L'aplatissement d'une donnée doit-il conserver le partage ? Typiquement un arbre binaire qui possède deux fils identiques (au sens de l'égalité physique) peut indiquer, pour le deuxième fils, qu'il a déjà sauvegardé le premier. Cette caractéristique influence la taille de la valeur sauvegardée et le temps mis pour le faire. D'autre part, en présence de valeurs physiquement modifiables, cela peut changer le comportement de cette valeur après une récupération selon que le partage a été conservé ou non.
- **valeurs circulaires.** Dans le cas d'une valeur circulaire, la linéarisation sans partage risque de boucler. Il sera nécessaire de conserver ce partage.
- **valeurs fonctionnelles.** Les valeurs fonctionnelles, ou fermetures, se composent d'une partie environnement et d'une partie code. La partie code correspond au point d'entrée (adresse) du code à exécuter. Que faut-il faire alors du code ? Il est possible de stocker uniquement cette adresse, mais alors seul le même programme trouvera un sens correct à cette adresse. Il est aussi possible de sauver la suite d'instructions machine de cette fonction, mais il sera nécessaire d'avoir un mécanisme de chargement dynamique de code.
- **garantir le type au réemploi** C'est la principale difficulté de ce mécanisme. Le typage statique garantit que les valeurs typées n'engendreront pas d'erreur de type à l'exécution. Mais cela n'est vrai que pour les valeurs appartenant au programme en cours d'exécution. Quel type peut-on donner à une valeur extérieure au programme, qui n'a donc pas été vue par le vérificateur de types ? Pour seulement vérifier que la valeur relue possède le type monomorphe engendré par le compilateur, il faudrait qu'il y ait une transmission de ce type au moment de la sauvegarde, puis vérification au chargement. Avec de surcroît un mécanisme de gestion de versions des types, pour rester sûr, même en cas de redéclaration d'un type dans le programme.

Pour son noyau fonctionnel et impératif, Objective Caml possède un mécanisme de «linéarisation» implanté par le module `Marshal` qui n'est pas sûr au niveau de la vérification de types et ne peut pas être utilisé pour les objets. De plus, dans le cas de sauvegarde de valeurs fonctionnelles il sera nécessaire de relire ces valeurs par le même programme pour être sûr que le pointeur de code des fermetures sont au même endroit. Un des intérêt d'utiliser la persistance objet est de communiquer des instances de classes entre différents programmes qui ont seulement la même définition de classe. On discute rapidement le problème de typage du module

`Marshal` pour ensuite présenter le nouveau module `DumpTo` qui opère de manière sûre sur les objets tout en montrant ses limitations.

module `Marshal`

Le mécanisme de linéarisation du module `Marshal` permet de conserver ou non le partage de valeurs. Il autorise aussi de manipuler des fermetures, dans ce dernier cas seul le pointeur de code est sauvé. Les deux fonctions principales sont :

- `to_string` : $\forall\alpha.\alpha \rightarrow extern\_flaglist \rightarrow string$ où α est un type objet
- `from_string` : $\forall\alpha.string \rightarrow int \rightarrow \alpha$

Le véritable danger avec ces fonctions est la possibilité de casser la sûreté de typage d'Objective Caml. Les types des fonctions de création sont monomorphes (*unit* ou *string*). Par contre les fonctions de dé-linéarisation ont un type polymorphe, *i.e.* le type du résultat est α et n'apparaît pas comme type des paramètres. Du point de vue des type, il y a une complète liberté de manipulation de valeurs persistantes. L'exemple suivant montre comment construire la fonction `magic_copy` de type $\forall\alpha\beta.\alpha \rightarrow \beta$.

copie

```
# let magic_copy a =
   let s = Marshal.to_string a [Marshal.Closures] in
    Marshal.from_string s 0;;
val magic_copy : 'a -> 'b = <fun>
```

L'utilisation d'une telle fonction transgresse le système de types et peut causer l'arrêt brutal de l'exécution du programme, et cela pour un problème de typage.

non sûr

```
#  (magic_copy 3 : float) +. 3.1;;
Segmentation fault
#  (magic_copy 3) +. 3.1;;
Segmentation fault
```

On perd alors le bénéfice du typage statique.

Persistance pour les objets

Pour remédier à cette situation on utilise l'extension `coca-ml` pour traiter les objets persistants. L'idée est de vérifier le type d'une valeur relue en effectuant un *cast* dynamique. Toutes les classes auto-sous-héritent d'une classe ancêtre `serialize`. Chaque objet relu est considéré comme du type de `serialize` et il sera alors possible d'effectuer un *downcast* vers sa classe originelle ou vers une classe intermédiaire entre celle-ci et `serialize`.

Un autre intérêt de faire de la persistance d'objets est de ne pas sauvegarder sa table des méthodes qui est partagée par toutes les instances d'une même classe. Ainsi seules les variables d'instance d'un objet sont à conserver. À la dé-linéarisation de créer un objet à partir des informations des variables d'instance en leur rattachant la table des méthodes de la classe d'origine.

Pour cela, chaque déclaration d'une classe concrète associe une clé de classe avec sa table des méthodes. En utilisant `coca-ml` on peut produire cette information, avec quelques restrictions, et les sauver dans une table de hachage.

Implantation Cette implantation a besoin de nouvelles actions pendant le *préprocessing* et de quelques fonctions C de bas niveau pour la sauvegarde des variables d'instances dans un flot d'octets. Le module `DumpTo` effectue ce lien entre ces fonctions C et un programme ML.

Pour chaque déclaration d'une classe concrète, le couple (`clé`, `TM`) est ajouté à la table de hachage pendant le préprocessing du programme source. Pour trouver la table des méthodes (`TM`), il est nécessaire de créer une fonction de construction d'instance de cette classe en effectuant une application partielle du constructeur `new`

sur le nom de la classe sans ses arguments. Dans l'exemple de la figure 2.12, la classe `left` a un argument pour son constructeur :

```
# let l = new left 3;;
l : left = <obj>
```

En cas d'application partielle de son constructeur, on obtient une fermeture qui contient dans son environnement la table des méthodes de la classe `left`.

```
# let al = new left;;
al : int -> left = <fun>
```

Dans ce cas on peut explorer l'environnement de cette fermeture pour trouver le pointeur de la table des méthodes.

Ainsi pour chaque déclaration concrète de classes, on ajoute à la table de hachage globale l'information de la clé de la classe et le pointeur de la table des méthodes.

Les fonctions C de bas niveau peuvent maintenant explorer un objet pour le linéariser. Le mécanisme de linéarisation conserve le partage de valeurs à l'intérieur des variables d'instance. Son format suit une syntaxe à la XML, principalement pour sa lisibilité. Un objet linéarisé contient toutes les variables d'instance (si elles ne sont pas fonctionnelles) et la clé de la classe.

Les deux fonctions principales du module `DumpTo` sont :

- `to_string` : $\forall\alpha.\alpha \rightarrow string$
- `from_string` : $string \rightarrow serialize$

et peuvent être utilisées de la manière suivante :

exemples

```
# let o = let s = DumpTo.to_string l in
     DumpTo.from_string  s;;
o : serialize = <obj>
# let t = cast o to top;;
t : top = <obj>
# let lt = cast o to left;;
lt : left = <obj>
# lt#eq(t);;
- : bool = true
# let r = cast o to right;;
Uncaugth exception : ...
```

Pour construire une clé unique pour une déclaration de classe, y compris entre deux programmes différents, on suit l'algorithme suivant :

- au premier niveau de déclaration, on crée une clé (*digest*) en utilisant l'algorithme MD5 sur son arbre de syntaxe ;
- pour les classes dérivées, on compose son *digest* créé à partir de l'arbre de syntaxe avec les "digests" hérités pour en créer un nouveau.

Limitations Il y a trois sortes de limitations :

1. choix d'implantation : les variables d'instance fonctionnelles ne sont pas traitées ;
2. restriction sur la structure de la déclaration de classe pour faciliter la recherche de la table des méthodes ;
3. les classes paramétrées ne sont pas traitées (restriction forte sur le polymorphisme paramétrique).

Dans le premier cas on ne peut pas garantir que le programme qui relit une valeur linéarisée possède le bon pointeur de fonction au bon endroit. Le deuxième cas interdit deux structures de déclarations de classes :

- les constructeurs sans argument car on ne peut plus faire une application partielle de `new` et parce que l'on crée alors un objet on peut changer le déroulement du programme en particulier parce que la méthode `initializer` est évaluée ;

– les variables de classe, simulées par la déclaration d'un environnement local, sont aussi interdites car la recherche de la table des méthodes n'est plus sûre.

Le troisième cas rejette les classes paramétrées car `coca-ml` ne sait pas garantir un «cast» sûr pour les classes paramétrées. Ces limitations peuvent paraître fortes, mais seul le troisième cas en restreint la portée du point de vue objet. Néanmoins `coca-ml` offre déjà un modèle objet permettant la construction de hiérarchie de classes à la Java tout en conservant le modèle d'Objective Caml.

Travaux connexes

Comme présentée à la section 4.3 la principale difficulté d'ajouter une vérification dynamique de type dans les dialectes ML est de devoir manipuler les types durant l'exécution. Par défaut, les informations de types sont perdues à l'exécution [8] pour réduire la taille des valeurs. Il y a seulement des étiquettes dans les valeurs pour aider le GC. Si l'on désire conserver une information de type complète, on retombe sur les difficultés de reconstruire le type exact d'une valeur comme par exemple :

```
let l = ref [];;
...
 l := [1;2];;
...
 l := []
```

Si l'information d'un type n'est pas dans la pile d'exécution (effet de bord, exception, application partielle) il devient obligatoire d'ajouter au moment d'une application une information partielle de type [3] pour reconstruire le type complet. Cette reconstruction peut aussi être utilisée pour le GC [72]. Une solution plus générale est d'introduire la notion de types dynamiques qui peuvent être testés par un filtrage de motif sur les types (`type-case`) [101]. Pour la programmation objet, le filtrage par types n'est pas la structure usuelle pour contraindre un type objet.

Pour la notion de persistance sûre (non objet) il est aussi possible d'analyser la structure d'une valeur pour déterminer si elle est compatible avec un type donné (*i.e.* cette valeur peut être utilisée sans risque comme une valeur de ce type), comme dans [63].

2.3 Extension parallèle : nocf

Caml-Flight est basé sur le modèle de programmation *SPMD* (*Simple Program Multiple Data*). Les communications sont explicites. Elles utilisent une primitive `get` pour l'évaluation distante d'une expression par un autre processus ou une autre machine. Ces communications sont autorisées uniquement à l'intérieur d'un bloc de synchronisation (`sync`). Ces blocs de synchronisation ne sont pas imbriqués. C'est une extension parallèle légère qui préserve la propriété de déterminisme de la partie fonctionnelle du langage.

La première implantation de Caml-Flight [62], principalement réalisée par Christian Foisy en 1994 était basée sur Caml-Light. Cette implantation modifiait profondément le compilateur Caml-Light en introduisant de nouvelles instructions pour le langage et sa machine virtuelle pour pouvoir exécuter plusieurs machines virtuelles dans le même processus. De plus un protocol complet au dessus d'UDP/IP pour supporter les évaluations distantes était construit ainsi qu'une forme de persistance pour la communication de valeurs. Une nouvelle version construite au dessus d'Objective Caml en 1998 suivait le même schéma de compilation en utilisant la bibliothèque de communication MPI pour la portabilité. Cette implantation a été utilisée sur des machines IBM SP2.

Une des principales difficultés pour maintenir ce type d'extension pour un langage est de rester à jour pour chaque nouvelle distribution du-dit langage. Chaque nouvelle version demande un nouveau *patch*. Caml-Flight n'a pas suivi les nombreuses évolutions d'Objective Caml.

Pour résoudre ce problème, on propose une nouvelle implantation, appelée Objective Caml Flight[9], qui ne modifie pas le compilateur original, mais fonctionne à la manière de `coca-ml` et de `scocaml` en compilant vers ML. Une telle méthode est maintenant possible parce que les nouvelles version d'Objective Caml introduisent des bibliothèques pour la concurrence, la répartition et la persistance. La grammaire de l'extension utilise encore une fois l'outil `camlp4`.

On présente tout d'abord l'extension Flight originale, pour ensuite décrire cette nouvelle implantation et les outils utilisés. On compare l'ancienne et la nouvelle implantation pour discuter les extensions parallèles du langage ML.

2.3.1 L'extension Flight

Comme Caml-Flight est un sur-ensemble de Caml-Light, on ne parlera que des ajouts et limitations par rapport à C des extensions apportées. On présentera l'adressage des processus, la syntaxe et le typage des nouvelles instructions, les environnements de communication et les blocs de synchronisation.

Adressage Les processus qui exécutent le calcul sont numérotés de `0` à `nodes - 1`. La valeur de `nodes` est commune à tous les processus. La constante `local` a la valeur de l'adresse du processus, cette valeur est unique et constante pour chacun des `nodes` processus.

L'intérêt d'un mode d'adressage linéaire est d'être indépendant de l'architecture du réseau ou de la machine. L'écriture d'applications portables en est facilité, au compilateur de s'adapter à la topologie du réseau ou de la machine cible.

Syntaxe Les deux constantes de localisation font partie de la nouvelle syntaxe. Deux nouvelles instructions sont ajoutées au langage C. Si on suppose qu'une expression C est nommée `Expr` et est fonction de `Simple_Expr`, l'extension de la syntaxe est la suivante :

`Expr ::= Simple_expr`
`  ...`
`  | sync Expr`
`  | get ( Expr , Expr )`
 	
`Simple_expr ::= ...`
`  | local | nodes`
	`get(e1,e2)` correspond à la demande de calcul, de l'expression `e2`, délocalisé sur le processeur d'adresse `e1`. `sync(e)` autorise les communications dans l'environnement (voir en 2.3.1) construit à l'entrée d'un `sync` pour le calcul de l'expression. Il n'y a pas de `get` en dehors d'un `sync`.

Typage Deux nouvelles règles de typage sont introduites pour les nouvelles instructions.

$\dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash sync(e) : \tau}$	Le type d'une opération parallèle `sync` d'une expression est le type de cette expression.

$\dfrac{\Gamma \vdash e_1 : int \; \Gamma \vdash e_2 : \tau \;\; V(\tau) = \phi}{\Gamma \vdash get(e_1, e_2) : \tau}$	Le type d'une communication est celui de l'expression à calculer. $V(\tau)$ est l'ensemble des variables de type de τ, il doit être vide (*i.e.* le `get` est monomorphe).

Les deux constantes de localisation, `local` et `nodes`, sont de type `int`.

[9] `http://www.pps.jussieu.fr/~emmanuel/Public/Dev`

Blocs de synchronisation, environnements et communications Il n'y a de communication possible qu'à l'intérieur d'un `sync`. À l'entrée du w-ième `sync` par un processus, l'environnement courant est conservé : il s'agit de l'*environnement de communication de la vague* w appelé EC par la suite. L'environnement de communication sert à l'évaluation des requêtes (produites par `get`) pouvant parvenir au processus. Pour cela, l'environnement de calcul ne pourra plus par la suite être modifié dans le corps de `sync` *avant un autre* `get`, *i.e.* il ne peut pas être augmenté, ni modifié sans interdire l'utilisation subséquente de `get`. Cela empêche toute nouvelle déclaration, le filtrage et (bien entendu) les modifications physiques avant l'occurrence d'un `get` dans le corps de `sync`. Cette restriction est particulièrement forte pour les filtrages à un seul cas.

Typiquement, le programme suivant est correct, mais sera rejeté par le compilateur Caml-Flight :

liaison interdite dans un sync

```
let f e =
  sync (
    match e with
     x -> if local < nodes - 1
          then get(local+1,x+y)
          else 0
  );;
```

En effet, le filtrage de l'expression `e` augmente l'environnement de calcul des variables `x` et `y`.

mais est indispensable pour la cohérence de l'environnement dans tous les autres cas.

Comme l'exécution des copies du programme est asynchrone, il est possible que certains processus soient entrés dans un `sync` donné et d'autres non. Si une requête est envoyée à un processus non synchronisé, alors celle-ci doit attendre que le processus en question arrive dans ce bloc de synchonisation. De plus, un processus ne peut communiquer avec un autre que si les deux évaluent le même `sync` à la même vague. Pour amoindrir les attentes et accepter des programmes avec décalage temporel, Caml-Flight autorise les calculs à des vagues différentes en introduisant le mécanisme de profondeur d'asynchronisme.

Profondeur d'asynchronisme Si on considère `sync` comme étant une barrière de synchronisation, tous les processus doivent s'attendre avant de débuter l'évaluation du corps du `sync`. Cette situation, essentielle dans les langages impératifs, est ici inutile puisque `get` accède un environnement composé essentiellement de valeurs (en excluant les valeurs mutables).

Pour relâcher cette contrainte, un paramètre entier est donné au lancement de l'évaluation : la profondeur d'asynchronisme. Si ce paramètre vaut p et que le processus le plus lent est à la vague w, le plus rapide ne pourra dépasser la vague $w + p$. Pour réaliser cette désynchronisation, les processus peuvent sauvegarder un total de $p + 1$ EC. De cette manière, un processus qui ne communique pas à une vague donnée peut continuer son évaluation plutôt que d'attendre à ne rien faire. On doit par contre connaître le moment où tous les processus ont terminé une vague donnée afin de libérer l'EC associé à cette vague, et ainsi permettre l'évolution du calcul. La profondeur d'asynchronisme agit ni plus ni moins comme une barrière de synchronisation à largeur variable : un processus trop «rapide» (ce qui arrive quand la charge est mal répartie) sera bloqué s'il atteint la vague maximale permise, son calcul ne pouvant reprendre qu'à la «mort» de la plus vielle vague.

La vague w correspond au w-ième bloc de synchronisation rencontré par un processus, il existe donc un environnement de communication par vague par processus. L'union des `nodes` EC d'une vague donnée forme l'environnement global de communication (EGC). L'EGC de la vague w doit être préservé tant que tous les processus n'ont pas atteint la vague $w + 1$.

La profondeur d'asynchronisme permet d'amoindrir les temps d'attente en permettant (quand le programme s'y prête) d'obtenir un effet pipeline entre les processus.

Exceptions et communications On ne s'intéresse qu'aux exceptions se déclenchant dans un `sync`. Deux cas sont possibles, soit l'exception est déclenchée localement, soit elle est déclenchée dans un `get` (*i.e.* sur un processus distant). Dans le premier cas, si l'exception est traitée dans le corps du `sync`, tout se passe comme

d'habitude. Si ce n'est pas le cas, l'exception s'échappe en provoquant prématurément la fin de la vague en cours . Si l'exception se produit à distance, dans la partie droite d'un `get`, celle-ci revient à l'appelant, qui déclenche celle-ci chez lui. L'exception rebrousse ainsi chemin (car un `get` peut en contenir un autre) jusqu'à ce qu'elle regagne le processus ayant initié la chaîne de `get`. On se retrouve ultimement dans le premier cas de l'exception locale.

Un exemple Le code suivant montre la syntaxe Caml-Flight. Pour le processus 0, l'appel de `f` retourne 0. Pour tous les autres processus, le résultat sera $2 * (local - 1) + 3$. Pour l'obtenir, il y a deux `get` qui demandent respectivement les valeurs $x + 1$ et $x + 2$ à leur voisin gauche.

exemple
```
let f x =
  sync (
    if Flight.local == 0 then 0
    else (get x + 4 from (Flight.local - 1)) (* GET 1 *)
       + (get x + 5 from (Flight.local - 1)) (* GET 2 *)
  );;

(f Flight.local);;
```

Récursion spatiale Un bloc de synchronisation `sync` ne peut pas apparaître à l'intérieur d'un autre `sync` (le parallélisme imbriqué est interdit), sauf durant une récursion spatiale où le `sync` imbriqué est le même que l'englobant. Dans ce cas l'évaluation reste à la même vague et l'environnement de communication reste inchangé.

Par exemple, la fonction `scan`[43] est autorisée :

scanf
```
let rec scanf (v: int list) =
  sync( if local == 0
        then [v]
        else v::(get(local-1,scanf v))) ;;
```

Ainsi durant la vague, des requêtes `get` sont demandées contenant un opérateur `sync`. Ce style de code est pratique pour transférer des données de n processus à 1. Dans cet exemple l'appel à `scanf` retourne une liste de listes d'entiers *int list list* de toutes les valeurs `v` du voisin gauche.

Limitations de Caml-Flight Un certain nombre de limitations existe dans l'implantation de Caml-Flight. La liste suivante les indique :

- **modifications physiques et communications** On ne s'intéresse qu'aux effets de bord effectués à l'intérieur d'un `sync` et en particulier en partie droite d'un `get`. Ceux-ci doivent être interdits parce qu'ils font perdre le déterminisme des programmes Caml-Flight. Pour le moment, le système de typage les accepte.
- **emboîtement des blocs de synchronisation** Actuellement, sauf pour les cas détectables syntaxiquement, l'emboîtement de `sync` est vérifié dynamiquement. Ce n'est pas dans l'esprit des langages de la famille ML.
- **get monomorphe** De même, la partie droite d'une opération communicante `get` est monomorphe. Cette contrainte est trop forte. Le problème de relâchement de cette contrainte s'apparente au typage des modifications physiques.

2.3.2 Nouvelle implantation : Objective Caml Flight

Notre contrainte principale est de rester compatible avec les nouvelles distributions d'Objective Caml. Pour cela nous ne modifions pas le compilateur originel. Un programme Flight sera traduit dans un programme Objective Caml utilisant une bibliothèque portable. Les outils suivants aident à cette tâche :

- l'outil `camlp4` pour l'extension de la grammaire ;
- la bibliothèque de *threads* pour exécuter concurrament les requêtes et l'évaluation courante ;
- la bibliothèque de *sockets* pour l'envoi de requête et le transfert des résultats ;
- la bibliothèque de persistance (`Marshal`) est utilisée pour geler les environnements de communication pour chaque `get` et pour retourner le résultat d'une requête.

L'extension de grammaires, le multi-threading, les prises de communication et la persistance sont de nouveaux traits du langage fournis dans les distributions récentes d'Objective Caml (la 3.06 est utilée ici). En utilisant ces outils, notre implantation actuelle tient en moins de 1000 lignes.

Description de l'implantation

Tous les processus peuvent tourner sur différentes machines. Ils communiquent à travers des *sockets* TCP/IP, et chaque processus connaît l'adresse IP et le numéro de port de tous les autres processus. Il y a une correspondance pour chaque adresse Objective Caml Flight (de 0 à $nodes - 1$) vers un couple (adresse IP, numéro de port). Parce que différentes instances d'un même programme peuvent tourner sur le même ordinateur, chaque d'eux possède un numéro de port différent des autres.

Instances d'un programme Chaque instance d'un programme connaît le nombre de processus (d'instances du programme) s'exécutant, sa propre adresse (`local`) comprise entre `0` et `nodes - 1` et pour chaque instance du programme : son numéro, son adresse IP et son numéro de port, ainsi que cette information pour tous les autres processus.

Une instance de programme est composée de deux *threads* : un *thread* pour l'exécution de son code séquentiel et un *thread* pour le serveur de requêtes qui répondra aux autres processus. Le protocole utilisé ici est TCP/IP. Pour chaque requête reçue un nouveau *thread* est lancé pour évaluer le calcul demandé. Quand ce calcul termine (ou est interromtpu par le déclenchement d'une exception) le résultat (ou l'exception) est envoyé à l'appelant.

Envoi d'une requête Une requête est l'évaluation de l'expression contenue dans un `get` dans un environnement de communications distant du même bloc de synchronisation (`sync`). Une requête est définie par le numéro de l'appelant (c), son numéro de vague (w) et son numéro de `get` (g). Quand ce triplet (w,s,g) est transmis alors le receveur peut démarrer un nouveau *thread*.

Réponse à une requête Le *thread* du serveur de requêtes doit calculer l'expression demandée à l'intérieur de l'environnement de communications correspondant au bloc `sync` indiqué dans la requête. Pour ce faire, au début de chaque bloc de synchronisation, toutes les requêtes potentielles (`get e from i`) de ce bloc sont gelées sous forme de fermetures (`fun () -> e`) et sont sauvegardées dans un tableau associé à ce bloc. Le numéro du `get` correspond à l'index de ce tableau. Quand une requête doit être calculée, un nouveau *thread* est lancé en appliquant la fermeture correspondante à un argument de type `unit` : `()`. Si cette application retourne un résultat, il est alors envoyé à l'appelant. Si une exception est déclenchée durant ce calcul, la valeur de l'exception est envoyée à l'appelant, celle-ci sera déclenchée localement. Ainsi chaque `sync` possède un tableau contenant les fermetures de tous les calculs des requêtes potentielles de son corps. Le numéro de `get` indique quelle fermeture utiliser. Le numéro du `sync` permet de vérifier si deux processus communiquant sont bien dans le même bloc de synchronisation à une vague donnée.

Pour diminuer les temps d'attente entre blocs de synchronisation, une profondeur d'asynchronisme est fixée au démarrage de tous les processus. Dans ce cas les p environnements de communications (tableaux de fermetures) sont conservés. Ainsi il reste possible d'appliquer une fermeture correspondant à un `sync` passé.

Environnement de communication L'environnement de communication d'un `sync` est créé à l'entrée du `sync`. Si un effet de bord est effectué dans le bloc ou en dehors avec une profondeur d'asynchronisme supérieure à 1 cet environnement peut changer. Dans ce cas la propriété de déterminisme est perdue. Une

analyse de flots n'est pas chose aisée à réaliser sans une modification du compilateur. Pour éviter cela chaque requête utilisera une copie de la fermeture correspondante, et donc de son environnement de calcul. Pour ce faire les environnements de communication sont sauvegardés sous forme de persistants en utilisant le module `Marshal`. La fonction `to_string` de type `'a -> string` sérialise les valeurs Objective Caml (à l'exception des objets), et la fonction `from_string` de type `string -> 'a` effectue l'opération inverse. Le tableau des fermetures devient alors un tableau de chaînes de caractères. L'environnement des fermetures ne peut pas être modifié sous cette forme. Quand une requête doit être traitée, la fermeture correspondant est alors dégelée. La valeur fonctionnelle ainsi obtenue possède une copie propre de son environnement. Par cette technique l'environnement de communication d'un `sync` ne pourra pas être modifié par un quelconque effet de bord.

Types des requêtes Le type des valeurs sérialisées par `Marshal` n'est pas conservé lors de la sérialisation, aussi la relecture n'est pas sûre pour le typage. Le type du résultat de la fonction `from_string` est `'a`. Pour ne pas casser l'exécution, le type de la valeur retournée doit être exactement celui attendu (inféré ou explicitement indiqué) au niveau du typage. Pour les fermetures ainsi gelées, on indique le type `unit -> 'a`, lequel est compatible pour toutes les fermetures. Ainsi il sera possible de les appliquer à `()` (valeur de type unit). Mais le type du résultat de cette application est inconnu (il ne peut pas être inféré à la compilation). Il y a un danger de manipuler son résultat, sauf dans notre cas, parce qu'à la fin de l'évaluation, le résultat sera transmis au processus appelant. Et cet appelant connaît le type attendu (type inféré lors du typage de l'expression contenue dans le `get`). Mais il reste un cas provoqué par le polymorphisme paramétrique où une même variable (d'un environnement de fermeture) peut être considérée de différents types. L'exemple suivant illustre ce cas :

get monomorphe

```
let f g x =
  let r = sync (
          if Flight.local = 0 then x
          else get x from (Flight.local - 1))
  in g x;;

let r = if Flight.local = 0 then f print_int 18
        else f print_string "Hello";;
```

La fonction globale `f` est polymorphe : `('a -> 'b) -> 'a -> 'b` et `r` a le type `unit`. Pour le processus 0, `x` est un entier et pour le processus 1 `x` est de type `string`. Quand le processus 1 envoie la requête `get x from 0`, le processus 0 retourne la valeur sérialisée 18. Malheureusement le processus 1 attendait une chaîne de caractères. Dans ce cas la discipline de types est cassée. Pour éviter ce cas, les expressions à l'intérieur d'un `get` doivent être monomorphes.

Récursion spatiale L'entrée normale dans un `sync` sauve l'environnement de communication à l'exception d'une récursion spatiale (même `sync` imbriqué).

Extension syntaxique

Comme pour `coca-ml` on utilise `camlp4` pour les nouvelles instructions du langage.

Les grammaires sont définies par un *lexer* et des entrées. Un *lexer* par défaut est fourni par la fonction `Plexer.make`. Les entrées sont définies dans les règles à l'intérieur d'une déclaration `EXTEND`. Chaque règle a une partie gauche décrivant la nouvelle syntaxe et une partie droite qui indique l'action associée.

On ajoute une nouvelle entrée, `sync_expr`, à la grammaire Objective Caml pour les expressions autorisées à l'intérieur d'un `sync` de la manière suivante :

```
value sync_expr = Grammar.Entry.create gram "expression";

EXTEND GLOBAL: sync_expr expr;
  expr :
    [[ "sync"; "("; c = sync_expr; ")" ->
       ...
    ]] ;
  sync_expr:
    [[ "get"; e = sync_expr; "from";  n = sync_expr ->
       ...
    ]];
```

La règle `sync` est au même niveau de priorité que la règle `expr` de la grammaire d'Objective Caml. Le corps d'un `sync` ne contiendra que des `sync_expr`. Cette nouvelle classe d'expressions est un sous-ensemble de l'entrée `expr` où les expressions introduisant de nouvelles variables sont absentes.

Le module `Flight`

Le module `Flight` gère les différents *threads*, le serveur de requêtes, la communication entre processus et définit le type des valeurs transférées. Chaque programme Objective Caml Flight doit finir son exécution par un appel à la fonction `close_Flight` pour attendre la fin de tous les autres processus. La gestion de la vague et de l'environnement de communication sont des zones mémoire à protéger, pour cela deux verrous d'exclusion mutuelle sont utilisés.

Le type suivant est défini pour encapsuler les valeurs transférées en incluant les exceptions.

type et exceptions
```
type 'a valeur = Valeur of 'a | Exn of exn;;
exception RemoteExn of exn;;
exception NestedSync;;
```

Une requête `get` sera traduite par le code suivant :

drequest
```
let drequest i n_get =
decode ( Marshal.from_string (request i n_wave n_sync n_get) 0)
```

où `request` demandera au processus `i` le `get` numéro `n_get` dans le `sync` numéro `n_sync` pour la vague numéro `n_wave`. La fonction `from_string` lit une valeur à partir de son format interne et la fonction `decode` filtre une `'a value`. Le corps d'un `sync` traite différemment le cas d'entrée normale d'une récursion spatiale.

Un exemple de traduction

La traduction suivante correspond au programme Objective Caml Flight donné page 58).

programme obtenu

```
let ___local_shift = !(Flight.num_sync)

let f x =
  let (___is_main, ___w_n, ___s_n, ___g_n,_) =
     Flight.get_hash_sync () in
  let ___fun_sync ___w_n ___s_n =
    if Flight.local == 0 then 0
    else
      (if Flight.local - 1 == Flight.local then x + 4
       else Flight.drequest (Flight.local - 1) ___w_n ___s_n 1) +
      (if Flight.local - 1 == Flight.local then x + 5
       else Flight.drequest (Flight.local - 1) ___w_n ___s_n 2) in
  if ___is_main then
    if ___s_n == 0 then
      Flight.reset_env_buffer ();
      Flight.save_closure (Obj.magic (fun () -> x + 10));
      Flight.save_closure (Obj.magic (fun () -> x + 20));
      Flight.begin_sync (___local_shift + 1);
      let ___res_sync =
        try ___fun_sync !(Flight.wave) (___local_shift + 1) with
          Flight.RemoteExn e -> Flight.end_sync(); raise e
        | e -> Flight.end_sync (); raise e
      in
        Flight.end_sync (); ___res_sync
    else if ___s_n == ___local_shift + 1 then
      ___fun_sync !(Flight.wave) (___local_shift + 1)
    else raise Flight.NestedSync
  else if ___s_n == ___local_shift + 1
       then ___fun_sync ___w_n ___s_n else raise Flight.NestedSync

let _ = f Flight.local
let _ = Flight.num_sync := !(Flight.num_sync) + 1
```

Le `sync` traduit sauve les deux requêtes potentielles et vérifie si c'est bien la première évaluation de ce `sync`. Le corps du `sync` est aussi gelé dans la fermeture `fun_sync`, principalement pour diminuer la taille du code. Chaque `get` est traduit par un appel à `drequest`. La valeur `is_main` indique le *thread* principal pour distinguer la première entrée et une récursion spatiale. Dans le premier cas l'environnement de communication est sauvegardé sous forme de fermetures par la fonction `save_closure` qui peuvent être sérialisées, pour autoriser des requêtes futures. Ces fermetures sont sérialisées par défaut. Dans le corps d'un `sync`, si la valeur de retour est une exception. Dans le deuxième cas, si le numéro du `sync` est différent une exception `NestedSync` est déclenchée. Au final la valeur globale `num_sync` est mise à jour.

Limitations

En comparaison avec la version originale, nous obtenons la même contrainte de types pour le `get` monomorphe et la même détection dynamique pour les `sync` imbriqués. De plus les `get` doivent être dans la portée lexicale d'un `sync`. Néanmoins la persistance d'Objective Caml autorise d'utiliser la partie impérative du langage tout en restant déterministe parce que quand le calcul d'une requête dégèle la fermeture correspondant à la requête il copie cette fermeture.

2.3.3 Performances et discussion

On présente ici quelques petits tests de performance et les comparons avec leur équivalent en Caml-Flight. Nous revenons sur les faiblesses du typage et discutons de l'intégration d'autres traits de programmation en Objective Caml Flight

Performances

On réutilise les programmes de test présentés dans [36] et [62]. La première table indique le comportement d'Objective Caml Flight dans les trois situations suivantes :

1. placement de processus dans un arbre d'appel d'appel récursif pour la fonction de `Fibonacci` (`fib_l`) ;
2. même calcul sur une structure linéaire de taille variable (`map`) ;
3. même calcul sur un vecteur (`wc_seek` et `life`) : le premier programme compte le nombre de mots pour un fichier donné. Le second calcule n étapes pour un jeu de la vie.

Tous ces programmes ont été exécutés sur un réseau local homogène (toutes les machines utilisées ont la même configuration matérielle et logicielle : Intel/Linux). Pour les programmes Objective Caml Flight on donne le temps réel du calcul.

Le facteur de vitesse est calculé de la manière suivante :

machine/OS	program name
Objective Caml	real (user+system)
Objective Caml Flight i/j/k	real
facteur	$\frac{(user+system)ObjectiveCaml}{(real)ObjectiveCamlFlight(i/j/k)}$

où i est le nombre de processus, j le nombre de processeurs et k la profondeur d'asynchronisme.

Intel/Linux	hello	fib_l	wc_seek	life
nombre de vagues	0 wave	1 wave	1 wave	30 waves
Objective Caml	0.3	7.7	6.7	13
1/1/0	0.85	10 **(0.77)**	9.5 **(0.7)**	19 **(0 .68)**
2/2/0	1.2	7.2 **(1,07)**	6 **(1.1)**	11.3 **(1.15)**
4/4/0	1.8	6 **(1.3)**	4.8 **(1.4)**	19 **(0 ,68)**
8/8/0	3.2	6 **(1.3)**	6.5 **(1)**	30 **(0, 43)**

FIG. 2.14 – Quelques tests

Si on compare avec les résultats anciens, il y a un coût plus important pour la construction des environnements de communication et pour le dégel des fermetures pour une requête.

Typage

Les deux implantations Caml-Flight et Objective Caml Flight ne sont pas sûres du point de vue typage pour le `get` polymorphe. Objective Caml suit la discipline de typage de [159] qui distingue les expressions en deux catégories : les expressions «expansives» (comme l'application), lesquelles sont dangereuses pour la généralisation d'un type en schéma de type, et les expressions «non-expansives» (comme les abstractions). Les variables de type d'une expression non-expansive peuvent être généralisées dans une déclaration `let`. Parce que nous travaillons par transformation de programmes, nous avons besoin d'ajouter du code pour transformer une expression incluant un `get` comme une expression expansive.

Le vérification dynamique des `sync` imbriqués n'est pas dans l'esprit de ML. Une analyse de flots, comme dans [120], pourrait être utilisée pour cette détection. Le cas du `get` monomorphe peut aussi être traité ainsi.

Autres traits de programmation

Objective Caml possède des traits de programmation de haut niveau comme les modules paramétrés, les hiérarchies de classes et les *threads*. Il serait intéressant de les intégrer dans Objective Caml Flight, mais cela pose des difficultés.

Les modules paramétrés (ou foncteurs) peuvent être appliqués à un ou plusieurs modules pour construire un nouveau module. Un foncteur peut être considéré comme une fermeture. Aussi il devient nécessaire de discriminer les différents blocs `sync` produits à chaque application d'un foncteur. La variable `___local_shift`, utilisées pour les modules, doit être ajoutées dans chaque déclaration d'un module paramétré. Une autre solution peut être de «linéariser» le code du foncteur sur les sites d'appel comme dans [139].

Pour l'extension objet, la principale limitation provient du module `Marshal` qui ne sérialise pas les objets dans la mesure où la table des méthodes d'une classe peut être déplacée lors d'un GC. Dans ce cas la reconstruction de l'objet n'est plus correcte et l'appel d'une méthode *crashe* le programme. Une solution est d'utiliser l'extension `coca-ml` qui en créant une hiérarchie de classes apporte une opération de «downcast» et un mécanisme de persistance. L'idée est alors de composer deux extensions réalisées en `camlp4`.

Le multi-threading fait perdre la propriété de déterminisme du langage. Par exemple si deux threads contenant un bloc `sync` s'exécutent concurremment, on ne peut pas déterminer lequel entre le premier dans le bloc `sync`. Dans ce cas, ce programme peut finir correctement ou bien être bloqué. Néanmoins le multi-threading peut aider à constuire de nouveaux modèles parallèles.

Discussion Ce prototype doit être vu comme un outil pédagogique : pour expérimenter le data-parallélisme en Objective Caml Flight et pour comprendre l'implantation d'extension parallèles utilisant les traits fonctionnels et impératifs du langage. Certaines de ces techniques peuvent être utilisées dans d'autres extensions parallèles comme BSML [77]. On obtient du point de vue implantation un haut niveau d'abstraction en utilisant la persistance des fermetures conjointement à la programmation *multi-threads*. `camlp4` peut cacher alors les transformations de programme manipulant ces traits.

Le moniteur associé n'est pas très puissant pour tracer ou profiler une exécution de programmes Objective Caml Flight d'où la nécessité de l'améliorer. En fait on a besoin de ce type d'outils pour d'autres extensions parallèles, comme par exemple BSML ou P3L[51].

Caml-Flight a toujours été considéré comme un langage de bas niveau. C'est probablement un bon langage cible. On espère que cette nouvelle implantation portable soit utiliséee pour faciliter la construction et les tests de nouvelles extensions parallèles.

2.4 Conclusion : extensions et typage

Les deux extensions, `cocaml` et `nocf`, et le compilateur `scocaml` utilisent principalement l'outil `camlp4`. Celui-ci permet une forme de macros en construisant à partir de règles d'analyse syntaxique un arbre syntaxique pouvant être typé, compilé et exécuté par Objective Caml. Cela y compris dans le cas de chargement dynamique comme le montre la fonction `_load` de `scocaml`. Néanmoins on s'aperçoit que ces trois exemples ont aussi besoin de manipuler une information de type :

- `scocaml` : l'affichage des types ne correspond pas aux types Scol mais à leurs traductions vers des types Objective Caml ;
- `cocaml` : les types objet sont reconnus par les noms des classes ;
- `nocf` : la contrainte du `get` monomorphe ne peut pas être exprimée.

Le mécanisme de quotation et antiquotation de `camlp4` est suffisamment riche pour les manipulations syntaxiques. Néanmoins dans le cadre d'un langage statiquement typé il devient nécessaire de pouvoir étendre et modifier le typeur en fonction des extensions. Dans le cas de `nocf` une solution pour l'opérateur `get` est de rendre une expression contenant un `get` comme «expansive» (au sens de Wright[159]) pour ne pas être généralisée, cela y compris à l'intérieur d'une abstraction. L'afficheur de types doit lui-aussi pouvoir être étendu/modifié pour intégrer le point de vue de l'extension au niveau des types.

Pour ce faire, la phase d'analyse syntaxique construit non seulement un AST mais ajoute des indications pour le typeur qui offre certains points d'accès dans la classification des expressions et pour l'affichage des types.

Chapitre 3

Interopérabilité

Le chapitre 1 a apporté des solutions techniques apportant un bon niveau de portabilité tout en étant efficace. Cela permet de répondre aux arguments anciens sur la lenteur d'exécution et la difficulté de déploiement. Le chapitre 2 a montré l'évolutivité intrinsèque du langage. Ce chapitre s'ouvre à d'autres langages (interfaces et interopérations). La possibilité et la facilité de communiquer avec d'autres langages sans une perte d'efficacité trop grande est un point important à nos yeux. Une application devient alors un assemblage de briques logicielles issues d'un ou de plusieurs langages.

L'assemblage de bibliothèques et/ou de composants pour la construction d'applications multi-langages est devenu une nécessité comme le rappelle l'introduction de [61] :

> «*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.* »

Certes les titres des articles «Sweet Harmony : the Talk/C++ connection» [53] et «Calling hell from heaven and heaven from hell» [60] montrent deux visions différentes. La manipulation de plusieurs langages peut selon leurs natures et les mécanismes d'interfaçage faire perdre certaines propriétés de sûreté de ceux-ci mais aussi bâtir de nouvelles constructions augmentant alors l'expressivité de ce mélange.

Les difficultés générales d'interfaçage entre langages sont les suivantes :

1. protocole d'appel des fonctions et des méthodes
2. conversion des types d'un langage à l'autre ;
3. copie ou partage des valeurs d'un monde à l'autre, en particulier la convention de passage de paramètres (in, out, inout) ;
4. ruptures de calcul (exceptions) transmises d'un monde à l'autre ;
5. gestion automatique de la mémoire (GC).

Ajoutons que certains traits de programmation d'un langage, qui ne se retrouvent pas forcément dans un autre langage, peuvent être difficilement traduisibles ou font perdre un facteur important d'efficacité. Ils peuvent apparaître au typage (selon les différentes classes de polymorphisme), sur les valeurs manipulées (comme les valeurs fonctionnelles) et sur le contrôle de l'exécution (exceptions, continuations).

La conception 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.
- intégrer dans un des langages un langage lisant les interfaces des modules/paquetages de l'autre langage ; les *stubs* (souches) peuvent alors être engendrés automatiquement ;

– utiliser un langage extérieur de description d'interface : IDL (Interface Description Language). De cette description le code peut être engendré automatiquement.

On présente tout d'abord ces trois choix pour l'interfaçage de CAML (Objective Caml et CeML) avec le langage C. On s'intéresse ensuite à la communication entre C et Objective Caml à travers l'interprète de la machine virtuelle d'Objective Caml.

Le souhait de pouvoir exécuter du code Objective Caml dans une application nous pousse à embarquer son *toplevel* pour permettre la compilation à la volée de sources. On décrit les modifications à apporter à la distribution Objective Caml pour ce faire. Des applications de cette technique seront discutées au chapitre 6. Ce travail est commun à Clément Capel, Jean-Marc Eber et moi-même.

On s'intéresse ensuite à l'interopérabilité d'Objective Caml avec le langage objet Java. Pour cela on présente un nouveau générateur de code, appelé `O'Jacaré`, pour faciliter l'interopérabilité entre Java et Objective Caml à travers leur modèle objet respectif. `O'Jacaré` définit un *IDL* simple permettant la description des classes et des interfaces afin de communiquer d'Objective Caml vers Java. Pour les communications de Java vers Objective Caml on ajoute un mécanisme de rappel. L'implantation repose sur les interfaces de bas niveau avec C de chaque langage (`Jni` pour Java et `external` pour Objective Caml) et utilise une version étendue de la bibliothèque `camljava`. `O'Jacaré` engendre toutes les classes encapsulantes nécessaires et vérifie le typage statique des deux mondes. Bien que l'*IDL* soit une intersection de ces deux modèles objet, `O'Jacaré` offre les caractéristiques des deux. Ce travail a été spécifié et implanté avec Grégoire Henry. Ses applications seront détaillées au chapitre 6.

Ces deux outils de communication nous permettent de mesurer l'évolution de l'interopérabilité entre CAML et d'autres langages, et de déterminer la prochaine étape.

3.1 Valeur, code et interface avec C

La manipulation dans un programme ML d'une valeur issue d'un autre langage ne va pas sans poser des difficultés principalement au niveau du typage et de la gestion mémoire. Les problèmes de gestion mémoire sont communs à l'ensemble des langages fonctionnels et aux langages munis de GC. La cohérence du typage des valeurs C par rapport au système de types du langage à interfacer est traitée différemment pour un langage typé statiquement comme ML ou typé dynamiquement comme Scheme. La problématique d'encapsulation de valeurs et de vérification de types est clairement exposée dans la thèse de Thierry Saura [133].

3.1.1 Encapsulation des valeurs

On suppose définie la structure `foo` suivante :

struct foo

```
struct foo { int a; struct bar { int y; char[ 4 ] s; } b; float c; }
```

La figure 3.1 montre la structure mémoire en ML ou Scheme pour intégrer une telle valeur C.

Un programme fonctionnel qui utilise cette structure `foo` devra fournir une capsule pour chaque instance de `foo`. Dans la figure 3.1, la capsule gauche pointe sur l'objet C. Si le même programme accède à la structure `bar`, il devra alors créer une capsule pour cette structure interne.

3.1.2 Appel au code externe

L'appel à du code C à partir de ML varie selon le langage cible du compilateur. Par exemple dans le cas de CeML la cible du compilateur est le langage C lui-même ce qui facilite le mécanisme d'appel (voir 2.3.2). Pour Objective Caml le code engendré par le compilateur natif suit le protocole d'appel de C ce qui 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 compilées vers le code-octet.

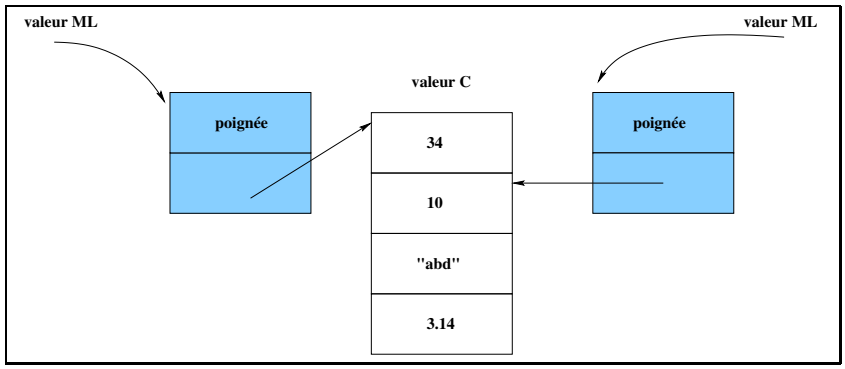


FIG. 3.1 – Représentations de la même donnée C en ML

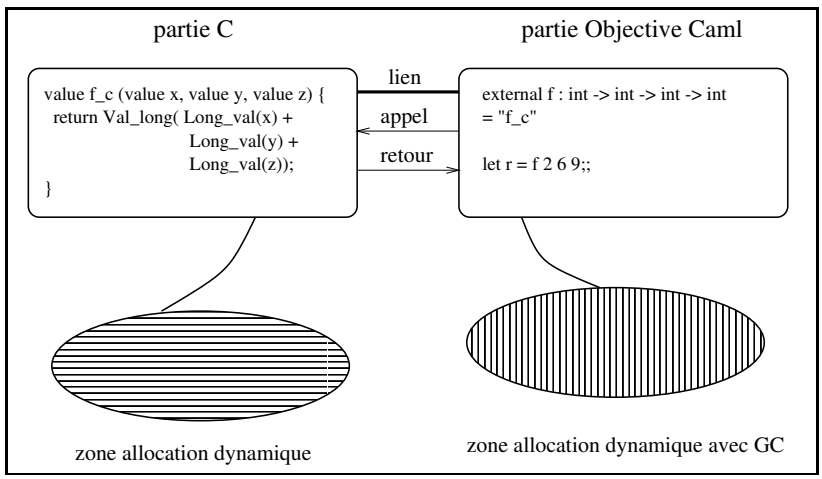


FIG. 3.2 – Communications entre Objective Caml et C

La figure 3.2 montre l'appel à la fonction `f_c` de C par un programme Objective Caml. Le lien est effectué par une déclaration `external` correspondant à une couche basse de l'interface externe de fonction ou *Foreign Function Interface* (*FFI*). La fonction C est vue du coté Objective Caml comme la fonction `f` de type `int->int->int->int`. Toutes les valeurs Objective Caml sont vues en C comme du type `value` (cela est du à la représentation uniforme). Le calcul en C nécessite de convertir en `int` C les paramètres `x`, `y` et `z` (macro `Long_val`). Une fois le calcul effectué en C, la valeur de retour est aussi convertie en valeur Objective Caml (macro `Val_long`). Dans cet exemple simple on ne se préoccupe pas de la gestion mémoire dans la mesure où les valeurs passsées d'Objective Caml vers C sont des valeurs immédiates. Si ce n'est pas le cas il est alors nécessaire d'ajouter les paramètres passés dans l'ensemble de racines du GC pour le temps du calcul. Il en est de même pour des allocations en C dans le tas Objective Caml.

structure d'un exécutable (code-octet) On utilise par la suite des variations du programme jouet suivant qui définit une fonction successeur :

succ.ml
```
let succ x = x + 1;;
let a = 2 in succ a;;
```

Lorsque l'on réalise l'édition de liens d'un tel programme[1], le compilateur de code-octet construit un fichier

[1] `ocamlc -o succ succ.ml`

exécutable possédant une structure particulière, comme décrit à la figure 3.3. L'ensemble des informations utiles à l'exécution du programme sont recensées. Parmi elles, on peut citer la table des symboles (qui stocke les noms des modules chargés, les exceptions ...) ainsi que la table des primitives (qui stocke les fonctions C utilisées par le programme).

RNTM	:	chemin du *runtime* (si il n'est pas concaténé)
CODE	:	instructions du programme
DLPT	:	chemins d'accès aux bibliothèques dynamiques
DLLS	:	bibliothèques dynamiques utilisées dans le programme
PRIM	:	liste des primitives utilisées dans le programme
DATA	:	données du programme
SYMB	:	table des symboles
DBUG	:	informations de *debug*
TOC	:	table des matières des différentes sections

FIG. 3.3 – Structure en section et table des matières de l'exécutable Objective Caml

communications à partir de C (code-octet) Par ailleurs, il est possible d'appeler une fermeture Objective Caml depuis C à travers l'interface externe (figure 3.4).

Dans ce cas l'édition de liens n'est pas réalisée de la façon précédente. En effet, pour être appelée en C, une fermeture Objective Caml doit avoir été stockée dans un fichier et compilée par le compilateur Objective Caml avec l'option `-output-obj`. Comme la cible devra être liée, ceci impose la génération d'un fichier C, rassemblant les mêmes informations qu'un exécutable Objective Caml dans un format valide pour un compilateur C.

f.ml
```
let f x = x + 1;;
Callback.register "f_caml" f;;
```
ocamlc -output-obj -o camlprog.o f.ml

main.c
```
value param = Val_long(2);
value v = callback(*caml_named_value("f_caml"), param);
int succ_2 = Long_val(v);
```
gcc -o main main.c camlprog.o -lcamlrun

FIG. 3.4 – Appel d'une fermeture Objective Caml en C / utilisation de l'option `-ouptut-obj`

Afin de stocker ces informations dans un fichier C, chaque section est convertie en un tableau C (figure 3.5). Toutefois, seules les sections correspondant au code, aux données et aux primitives sont conservées dans le programme C.

a.o.c
```
static int caml_code[] = {0x00000054, 0x00000390, ...};
static char caml_data[] = {132, 149, 166, 190, ...};
primitive builtin_cprim[] = { array_get, array_set_addr, ... };
char * names_of_builtin_cprim[] = "array_get","array_set_addr", ... };
void caml_startup(char** argv){...}
```

FIG. 3.5 – Fichier C engendré par `-output-obj` qui sera lié au code C existant

Lors de l'exécution du programme C, l'appel à la fonction `caml_startup` de la figure 3.5, située dans le fichier engendré par `-output-obj`, permet d'appeler la fonction d'initialisation de la machine virtuelle

avec les tableaux de code, de données et de primitives. Les fermetures sont alors ajoutées à l'environnement d'exécution, et les fonctions exportables vers C recensées.

3.1.3 Interface

Comme indiqué dans l'introduction de ce chapitre, une interface entre deux langages doit tenir compte du système de types, de la gestion mémoire (partage ou copie, passage d'arguments, GC) et du contrôle (appel de fonctions ou méthodes et rupture de calcul). À cela on peut ajouter le *multi-threading* quand un ou les deux langages possèdent cette caractéristique. Vu la complexité de la tâche, l'automatisation d'un certain nombre d'étapes est souhaitée. Dans tous les cas l'interfaçage repose sur une communication de bas niveau. Si deux langages possèdent une interface avec C ou avec le même code-octet, il devient possible alors d'interfacer directement ces deux langages en masquant le passage vers le C ou le code-octet. Là aussi l'automatisation de la génération des codes intermédiaires est souhaitable.

interface manuelle

Pour illustrer le propos précédent, on décrit un petit programme Objective Caml appelant une fonction C qui elle-même appelle une fermeture CAML. Le fait d'avoir une double communication nécessite d'être soigneux dans la gestion mémoire pour éviter qu'un déclenchement de GC en Objective Caml ne libère une valeur encore utilisée en C.

avec C Après avoir défini la fonction globale `plus`, Objective Caml enregistre la fermeture `(plus 3)` pour C.

```
# let plus x y = x + y ;;
val plus : int -> int -> int = <fun>
# Callback.register "plus3_ocaml" (plus 3);;
- : unit = ()
```

La fonction `plus3_C` appelle la fermeture Objective Caml `plus 3`. Il est à noter que le paramètre transmis `v` et la fermeture `f` doivent être ajoutés à l'ensemble des racines du GC d'Objective Caml par les macros `CAMLparam1` et `CAMLlocal`. Ces racines seront libérées par la macro `CAMLreturn`.

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/callback.h>

value plus3_C (value v)
{
  CAMLparam1(v);
  CAMLlocal1(f);
  f = * caml_named_value("plus3_ocaml") ;
  CAMLreturn callback(f,v) ;
}
```

Enfin Objective Caml déclare `external` la fonction `plus3_C` en la nommant `plusC`, puis l'appelle.

```
# external plusC : int -> int = "plus3_C" ;;
external plusC : int -> int = "plus3_C"
# plusC 1 ;;
- : int = 4
# Callback.register "plus3_ocaml" (plus 5);;
- : unit = ()
# plusC 1 ;;
- : int = 6
```

Il n'y a pas exactement correspondance entre la déclaration d'une primitive C par `external` et l'enregistrement d'une fermeture Objective Caml par la fonction `register`. Dans le premier cas, il s'agit d'une déclaration statique, le lien entre les deux noms se fait à l'édition de lien. Alors que dans l'autre cas, il s'agit d'une liaison dynamique qui intervient durant l'exécution. En particulier, rien n'interdit de remplacer la liaison nom–fermeture en remplaçant la fermeture et en modifiant ainsi le comportement des fonctions C utilisant ce nom.

avec C# Dans le cadre de la compilation vers un même *runtime* possédant un GC, à la manière de Java ou de .NET, il n'y a plus besoin pour l'interface de bas niveau de se soucier de la gestion mémoire. Nous y reviendrons à la fin de ce chapitre.

langage d'interface intégré au langage

Dans le but d'automatiser la construction de valeurs et l'appel de fonctions C, une extension de CeML a été réalisée en choisissant de prendre comme langage d'interface, les 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 ML pour les manipuler. Ces nouveaux types sont considérés abstraits par le typeur ML.

```
#extern "include <\"sys/time.h\">";;
#extern "include <\"sys/resource.h\">";;
#extern "typedef timev  =
           struct timeval {
             long    tv_sec;
             long    tv_usec;
           } ctype1;";;
#extern "typedef rusage =
           struct rusage {
                struct timeval ru_utime;
                struct timeval ru_stime;
                ...
                int  ru_nivcsw;
           } ctype3;" ;;
#extern "type RU =
                struct rusage *ctype4;";;
#extern "let_macro RUSAGE_SELF =
                int RUSAGE_SELF;";;
#extern "let getrusage =
           int getrusage(int who,
              struct rusage * irusage);";;
```

```
let print_timeval t =
  print_string "{tv_sec=";
    (access_struct_timeval_tv_sec t);
  print_string ";tv_usec=";
  print_int
    (access_struct_timeval_tv_usec t);
  print_string "}";;

let r = create_star_struct_rusage ();;
let i = getrusage RUSAGE_SELF r;;
print_timeval
  (access_struct_rusage_ru_utime
     (access_star_struct_rusage r));;

...

let i2 = getrusage RUSAGE_SELF r;;
print_timeval
  (access_struct_rusage_ru_utime
     (access_star_struct_rusage r));;
```

FIG. 3.6 – Interface et programme CeML

Dans la figure 3.6, la déclaration `timeval` crée le type abstrait `struct_timeval`, une fonction de création `create_struct_timeval` de type `unit -> struct_timeval` et, dans ce cas, des fonctions d'accès et de modification pour chaque champ de la structure :

```
access_struct_timeval_tv_sec : (struct_timeval -> int)
modify_struct_timeval_tv_sec : (struct_timeval -> int -> unit)
```

Toutes ces fonctions sont monomorphes, y compris celles d'accès à un pointeur ou à un tableau. Les interfaces pour les macros et les variables nécessitent de donner le nom et le type C ainsi une nouvelle variable ML est créée avec le type ML correspondant. Comme résultat de la correspondance directe des fonctions C et ML, il est possible d'appliquer partiellement une fonction C en passant par cette interface.

langage de description d'interface autonome

L'utilisation de langages de définitions d'interface (*IDL*) permet de décrire les signatures des fonctions, procédures, 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. 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[138] défini par l'OMG[2] 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). Souvent ils sont construits pour des besoins d'interopérabilité dans la répartition comme pour DCE[160] de l'*Open Group*[3]. La troisième grande famille d'*IDL* est orientée pour l'interfaçage des composants COM de Microsoft[4].

Les interfaces externes des langages fonctionnels sont passées de liens de bas niveau avec C et C++ [53], [60] à des *IDL* pour interfacer les composants COM (H/Direct [61] et Caml*IDL*[5]). Toutes ces interfaces doivent gérer l'interaction des différents gestionnaires de mémoire, d'exceptions et de *threads* le plus souvent en passant par C.

Ces langages de définitions d'interface sont en règle générale complexes parce qu'ils veulent intégrer de nombreuses fonctionnalités. Nous présentons à la section 3.3 un *IDL* simple et néanmoins riche pour interfacer Objective Caml avec Java *via* leurs modèles objets respectifs.

3.2 Toplevel embarqué

Il n'est pas rare qu'un logiciel ait besoin au cours de son exécution d'évaluer des fragments de programmes issus d'un autre langage que celui de son implantation. On parle alors de langages de «glu», de «macros», de «scripts» ou de langages embarqués.

Cela est particulièrement utile pour la paramétrisation de l'application mais aussi pour l'interaction avec l'utilisateur. On peut citer un certain nombre de langages de script comme sh pour Unix, Lisp pour `Emacs` et `Gimp`, Visual Basic dans Excel, javascript pour les page Web du côté client et Php ou perl du coté serveur, Lua[6] ou VSL[7] pour les jeux vidéo, etc. Dans d'autres cas, c'est le programme lui-même qui engendrera le texte source à évaluer, par exemple pour la construction d'une requête SQL. Ces langages sont plus faciles d'utilisation que l'emploi de la bibliothèque de l'application tout en ayant accès à l'environnement de celle-ci. Ils sont le plus souvent interprétés. L'interpréteur embarqué dans une application est lié aux fonctions de celle-ci tout en sachant communiquer des valeurs de son format à celui de l'application. La genèse de ces langages répond souvent à un besoin immédiat. En cas de succès, ces langages ont tendance à devenir plus importants en incorporant de nouvelles fonctionnalités. Il en est de même des programmes écrits avec ceux-ci qui, souvent, grossissent fortement. On rencontre alors des difficultés liées à des sémantiques alourdies par la compatibilité ascendante et à de piètres efficacités dues en partie à l'interprétation. Par exemple le programme `latex2html`[8] écrit en perl dépassait les dix mille lignes et avait de réelles lenteurs et difficultés pour traduire des documents importants. Certains de ces langages sont compilés vers du code-octet. Celui-ci peut être ensuite interprété par une machine virtuelle embarquée (Lua, VSL). L'intérêt est alors un gain d'efficacité à l'exécution.

Le mécanisme d'embarquement d'un langage est proche de celui de la boucle d'interaction (*toplevel* ou *Read-Eval-Print Loop*). La partie lecture peut être interactive ou non ; la partie évaluation est quasiment identique ; la partie affichage devient la transmission d'une donnée dans un format accessible. Comme l'évaluation d'une

[2] `www.omg.org/`
[3] `www.opengroup.org`
[4] `www.microsoft.com`
[5] `caml.inria.fr/camlIDL`
[6] `www.lua.org`
[7] `www.virtools.com`
[8] `www.latex2html.org/`

phrase peut provoquer une erreur, celle-ci doit être détectable et manipulable dans l'application. La nature du typage du langage (non typé, typé dynamiquement, typé statiquement) influe sur la détection de certaines erreurs. Dans le cas de code compilé le type de chargement (statique ou dynamique) du code influence aussi l'intégration dans une application. Enfin un dernier critère est celui de légèreté de l'embarquement.

Il n'est pas surprenant de retrouver l'ancêtre des langages fonctionnels comme langage de macros dans des applications bien connues comme Emacs et Gimp. La tradition de la boucle *Read-Eval-Print* vient des premiers interprètes Lisp. De plus ce langage d'une clarté agréable a un pouvoir d'expression impressionnant. Même un langage comme javascript s'en inspire avec la construction `function`. La vague Java a proposé d'embarquer une machine virtuelle munie d'un mécanisme de chargement dynamique des classes. Mais il manque la possibilité d'exécuter des programmes Java source. Rien n'empêche d'embarquer un interprète Java comme «dynamic Java»[9] et même un compilateur Java comme «janino»[10] autorisant ainsi l'exécution à la volée de textes Java. L'expérience du langage Scol[11][23] montre les possibilités de chargement dynamique dans un langage statiquement typé. La plate-forme .NET [12], qui se veut un creuset où s'intègrent différents langages, possède dans son *API Reflection* les méthodes pour engendrer et charger dynamiquement du code-octet.

L'importance que peuvent prendre ces langages dans une application nécessite des précautions lors de leur conception. On rejoint alors les problématiques d'expressivité, de sémantique claire et de compilation efficace des langages de programmation. Pourquoi alors ne pas choisir un langage existant ayant ces bonnes caractéristiques ? Nous nous intéressons à l'utilisation du langage Objective Caml dans ce cadre.

3.2.1 Manipulation du *toplevel*

Bien qu'une bibliothèque Objective Caml puisse être utilisée dans un programme C il n'est pas possible d'y embarquer le *toplevel*.

manipulation du *toplevel* par un programme Objective Caml Comme le *toplevel* Objective Caml est entièrement écrit en Objective Caml, il est possible de l'appeler au sein même d'un programme Objective Caml en le liant avec la bibliothèque `toplevellib.cma`. Ceci permet alors d'évaluer du code créé à l'exécution par le programme Objective Caml lui-même. Il est ainsi possible de définir une fonction `eval` (figure 3.7) pour évaluer des textes Objective Caml (expressions ou déclarations).

maniptop.ml

```
Toploop.initialize_toplevel_env();;
let eval txt = let lb = (Lexing.from_string txt) in
  let phr = !Toploop.parse_toplevel_phrase lb in
  Toploop.execute_phrase true Format.std_formatter phr;;

eval "let add1 x = x +1;;";;
eval "add1 2;;";;
```

chargement dans le *toplevel*

```
# #use "maniptop.ml";;
- : unit = ()
val eval : string -> bool = <fun>
val add1 : int -> int = <fun>
- : bool = true
- : int = 3
- : bool = true
```

édition de liens

```
ocamlc -custom toplevellib.cma maniptop.ml -o manip
```

exécution

```
> ./manip
val add1 : int -> int = <fun>
- : int = 3
```

FIG. 3.7 – Le *toplevel* manipulé depuis un programme Objective Caml

[9] `koala.ilog.fr/djava`
[10] `www.janino.net`
[11] `www.scol-technologies.org`
[12] `www.microsoft.com/net`

impossibilité de manipuler le *toplevel* en C Il n'est pas malheureusement pas possible de manipuler d'exécuter le programme `maniptop.ml` en C. Lors de son initialisation (fonction `Symtable.init_toplevel`) le *toplevel* Objective Caml doit initialiser l'environnement du compilateur (écrit en Objective Caml) avec la liste des primitives et la table des symboles. De fait, comme la table des symboles n'est pas disponible à l'exécution (figure 3.5), toute tentative d'appeler le *toplevel* Objective Caml en C conduit à une exception. En outre, si les primitives sont disponibles dans l'environnement de la machine virtuelle, elles ne le sont pas dans celui du compilateur, rendant indisponible la compilation de fermeture liée à du code C et donc une grande partie de la bibliothèque standard.

3.2.2 Création d'une bibliothèque *toplevel*

Le but est de rendre disponible le *toplevel* sous forme d'une bibliothèque en C. Des fonctions d'appel simples sont donc proposées pour permettre de communiquer avec la boucle d'interaction, qui s'exécute bien sûr dans le même processus que le programme C appelant. L'utilisation est identique au *toplevel* classique et autorise toutes directives, déclarations et options disponibles habituellement. Deux fonctions principales sont fournies :

- appel préalable de `toplevel_init` (1 seule fois) ;
- appels de `toplevel_exec` (à chaque exécution d'une phrase) ;

et l'édition de liens s'effectuera par un ordre de la forme suivante :

```
gcc -o nom_executable fichiers_c -ltoplevel
```

ajout de la table des symboles Pour cela, il s'agit donc, d'une part de stocker la table des symboles dans le fichier engendré par `-output-obj` et d'autre part de transmettre les informations de primitives et de table des symboles de la machine virtuelle (en C) au compilateur (en Objective Caml). Pour obtenir une cohérence du fichier généré par `-output-obj`, la table des symboles doit être écrite dans un format C. On utilise alors le module `Marshal` pour ranger la table des symboles dans un tableau C d'entiers.

ajout des primitives Les primitives sont en fait des fonctions C qui ont la particularité d'être appelables par du code Objective Caml. Elles font partie du runtime `ocamlrun`[13] (figure 3.8 (a)). Afin de pouvoir être exécutées par l'interprète (figure 3.8 (b)), elles bénéficient d'un traitement spécial pour être connues à la fois de la machine virtuelle et du code interprété (figure 3.8 (c))

Dans notre cas, il faut que la table des symboles soit accessible du code qui va être exécuté (fonction d'initialisation du *toplevel*). On a donc créé une primitive pour réaliser ce transfert. En outre, comme la liste des primitives est aussi récupérée par `init_toplevel` (figure 3.8 (d)), elle doit subir le même type de transfert.

Une primitive est donc ajoutée, `get_symtable`, renvoyant une chaîne de caractères Objective Caml. Cette chaîne est en fait la linéarisation de la table des symboles contenue dans le fichier engendré par `"-output-obj"`, table transmise lors de l'initialisation du *runtime*. Pour que le programme Objective Caml accède à cette primitive, une interface est déclarée, se chargeant de l'appel de primitive (figure 3.8 (e)).

Il suffit alors au programme Objective Caml d'appeler la fonction de l'interface pour récupérer cette chaîne. Afin de récupérer la liste des primitives (figure 3.8 (f)), le même traitement est réalisé. Il est alors possible d'initialiser le *toplevel* avec toutes les informations nécessaires.

chargement dynamique Cependant, cela ne suffit pas pour réaliser tout chargement dynamique dans le *toplevel* embarqué. En effet, le *runtime* ne considère pas le cas d'un *toplevel* Objective Caml appelé depuis C. De ce fait, la table des primitives du runtime est initialisée de façon complètement statique, il n'est donc pas possible de rajouter des primitives C (cas de la bibliothèque graphique). Pour cela, il faut donc réécrire

[13] elles peuvent aussi être dans une bibliothèque dynamique liée à une bibliothèque Objective Caml (`threads.cma`, `graphics.cma`).

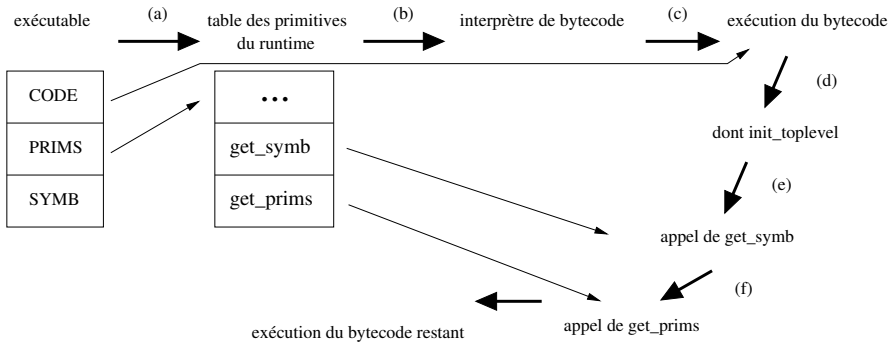


FIG. 3.8 – Fonctionnement de l'appel de primitive

l'initialisation de cette table en utilisant les fonctions d'accès, tout en maintenant la cohérence de la table lorsqu'elle est enrichie.

interface Afin, de pouvoir manipuler ce *toplevel* embarqué de façon simple, une interface est fournie.

Du côté Objective Caml, l'initialisation prend en charge l'analyse des arguments, de la même façon que l'exécutable `ocaml`. Il est donc par exemple possible de préciser des chemins de chargement et/ou des fichiers (sources ou compilés) à charger pour définir l'environnement initial d'exécution.

Du côté C, outre la gestion du passage d'argument entre C et Objective Caml, l'environnement de la console est pris en charge. Ceci permet en particulier d'utiliser les *threads*, qui vont faire passer la console en mode non-bloquant, la bibliothèque se chargeant de repasser en mode bloquant afin de permettre une éventuelle saisie au clavier en C.

Afin d'assurer le maximum d'expressivité, 3 fonctions d'exécutions sont proposées :

- `char* schema_toplevel_exec(char* code)` : retourne la sortie classique du *toplevel* (type + valeur) ;
- `value toplevel_exec(char* code)` : retourne la valeur de l'expression évaluée (ou une exception Objective Caml) ;
- `VARIANT com_toplevel_exec(char* code)` : retoune un variant COM.

3.2.3 Exemple

Il est maintenant possible de manipuler le *toplevel* Objective Caml en C. Voici un exemple simple de calcul de la factorielle par ce *toplevel* embarqué :

programme C : ex.c

```
toplevel_init(argv);

toplevel_exec("let rec fac x = match x with 0 -> 1 | 1 -> 1 | x -> x*fac (x-1);;");
val = toplevel_exec("fac 5;;");
res = 1+Long_val(val);
printf ("1+5!=%d\n", res);
```

La compilation suivante construit l'exécutable `ex.exe`.

édition de liens

```
gcc ex.c -o ex.exe -ltoplevel
```

Et l'exécution de `ex.exe` affiche bien le résultat attendu.

exécution

```
1+5!=121
```

Des applications utilisant ce *toplevel* embarqué, en particulier l'exécution d'applications graphiques dans un navigateur Web, sont présentées au chapitre 6.

3.3 IDL pour Java et Objective Caml

JAVA [75] possède un mécanisme d'interfaçage avec C nommé JNI [74] (Java Native Interface). Objective Caml comme nous l'avons vu a lui-aussi une interface externe avec C. Ces deux langages peuvent donc potentiellement communiquer entre eux *via* C. L'outil `camljava`[14][99] fournit cette interface de bas niveau entre les deux langages. Néanmoins comme toute interface de bas niveau `camljava` est difficile et non sûr d'emploi du point de vue des types. Par contre il intègre déjà les mécanismes de coopération entre les deux GC. On cherche alors à manipuler les objets d'un langage par l'autre de manière simple et sûre. Comme ces deux langage ont des modèles objets qui diffèrent, on présente un petit IDL correspondant à l'intersection des deux modèles objets pour simplifier cette interaction. La génération automatique des codes Objective Caml et Java garantit le typage fort d'Objective Caml et autorise une forme de *downcast* pour la partie Java manipulée en Objective Caml.

comparaison Java - Objective Caml

caractéristiques	Java	Objective Caml	caractéristiques	Java	Objective Caml
classes	√	√	sous-typage	√	√
accès champs	1	2	héritage ≡ sous-typage ?	oui	non
liaison tardive	√	√	surcharge	√	5
liaison précoce	√	3	héritage multiple	6	√
typage statique	√	√	classes paramétrées	7	√
typage dynamique	√	4	paquetages/modules	8	8
			classes mutuellement récursives	√	9

Un √ indique que le langage supporte un trait de programmation quitte à en préciser la nature par une note numérotée ci-dessous.

1. selon la nature des attributs de visibilités les variables de classes ou d'instances sont accessibles ;
2. uniquement *via* un appel de méthode ;
3. les méthodes statiques sont des fonctions globales d'un module et les variables de classes des déclaration globales ;
4. pas de *downcast* en Objective Caml seulement dans l'extension coca-ml [33] ;
5. pas de surcharge en Objective Caml mais le type de `self` peut aparaître dans le type d'une méthode qui pourra être redéfinie (*overriding*) dans une sous-classe ;
6. pas d'héritage multiple pour les classes Java, seulement pour les interfaces ;
7. pas de classes paramétrées en Java, seulement dans les extensions Pizza [155] et Generic Java [26] ;
8. les modules simples d'Objective Caml correspondent aux parties publiques des paquetages Java ; la notion de modules paramétrés est inexistante en Java ;
9. avec une limitation car les modules ne sont pas mutuellement récursifs.

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. C'est effectivement un modèle objet simple pour un langage, mais il va nous servir pour la communication entre des classes Java et Objective Caml.

[14] `caml.inria.fr/camljava`

3.3.1 Description de l'IDL

Cet IDL est orienté pour interfacer Java pour Objective Caml. Pour cela il s'inspire fortement de la syntaxe Java et, bien qu'il privilégie le sens d'appel d'Objective Caml vers Java, un mécanisme de *callback* autorise la redéfinition d'une méthode Java par une méthode Objective Caml et l'implémentation en Objective Caml d'une interface Java.

Principes

Notre motivation principale étant la simplicité d'emploi, l'IDL défini n'accepte que les déclarations de classe, de classe abstraite et d'interface (voir figure 3.9) dans une série d'espace de noms. Celles-ci correspondent respectivement à une classe, une classe abstraite et une interface Java dans une série de package, et à une classe ou une classe abstraite Objective Caml encapsulant un objet Java dans les deux derniers cas. L'héritage suit celui de Java : simple pour les classes mais multiple pour les interfaces. Du point de vue du programmeur un appel depuis Objective Caml vers Java se fait donc par un appel de méthode, et le sens inverse s'effectue aussi par un appel à une méthode *via* un mécanisme de *callback*. Les variables d'instance ou de classe Java étant accessibles par des méthodes engendrées automatiquement depuis Objective Caml. Seules les valeurs des types de base sont donc passées (ou retournées) par copie, dans tous les autres cas (objet, tableau) les valeurs sont transmises par référence assurant ainsi le partage. On distingue dans l'IDL le type de base `string` d'un objet de type `java.lang.String`. Les exceptions sont propagées d'un monde à l'autre. Le code engendré à partir d'un fichier interface est sûr du point du typage : la cohérence de l'IDL avec les classes Java étant vérifié à l'initialisation du programme. Il en est de même pour la gestion mémoire où deux GC cohabitent : le GC d'Objective Caml prenant en charge les objets Java dont il demande l'allocation.

Syntaxe

La syntaxe du langage d'interface est donnée à la figure 3.9. On utilise les accolades ({...}) pour les règles optionnelles pouvant être vides ([...]*). Les espaces de noms sont indiqués par le mot clé **package**.

classes et interfaces	arguments, attributs, types
file : := *package* { *package* } | *decl* { *decl* } *package* : := [**package** *qname* ;] *decl* { *decl* } *decl* : := *class* | *interface* *class* : := [[*attributes*]] [**abstract**] **class** *name* [**extends** *qname*] [**implements** *qname* { , *qname* }] { { *class_elt* ; } } *class_elt* : := [[*attributes*]] { **static** | **final** } *type name* | [[*attributes*]] { **static** | **abstract** } *type name* ([*args*]) | [[*attributes*]] < **init** > ([*args*]) *interface* : := [[*attributes*]] **interface** *name* [**implements** *qname* { , *qname* }] { { *interface_elt* ; } } *interface_elt* : := [[*attributes*]] *type name* | [[*attributes*]] *type name* ([*args*])	*args* : := *arg* { , *arg* } *arg* : := [[*attributes*]] *type* [*name*] *attributes* : := *attribute* { , *attribute* } *attribute* : := **name** *ident* | **callback** | *type_attribute* *type_attribute* : := basetype | *object* | *type_attribute* **array** *type* : := *basetype* | *object* | *basetype* [ ] *basetype* : := **boolean** | **byte** | **char** | **short** | **int** | **long** | **float** | **double** | **string** | **top** *object* : := *qname* *qname* : := *name* { . *name* }

FIG. 3.9 – Grammaire de l'idl

Les constructeurs sont nommés <**init**>. Les attributs de classe, de méthodes ou de variables d'instance concernent les alias de nom, les conversions explicites de types et la qualification pour le mécanisme de **callback**. Les types manipulés sont les classes, les tableaux et les types de base auxquels ont été ajoutés **string**.

Exemple d'utilisation

```
package mypack;

class Point {
  int x;
  int y;

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

  void moveto(int,int);
  void rmoveto(int,int);
  string toString();
  void display();
  double distance();
  boolean eq(Point);
}
```

```
[callback] class PointColore extends Point {
  [name default_point_colore] <init> ();
  [name make_pc] <init>
     (int,int,string);

  string getColor();
  void setColor(string);
  bool [name eq_pc] eq(PointColore)
}

class Nuage {
  [name empty_nuage] <init> ();

  void addPoint(Point);
  string toString();
}
```

FIG. 3.10 – Les inévitables classes `Point` et `PointColore`.

Dans cet exemple on cherche à utiliser depuis Objective Caml les classes `Point` et `PointColore` écrites en Java en sachant que la méthode `toString` de la classe `PointColore` retourne la concaténation d'un appel à la méthode `toString` sur `super` et d'un appel à la méthode `getColor` sur `this`.

La compilation du fichier `p.idl`, figure 3.10, engendre d'une part un fichier Objective Caml (`p.ml`) contenant le code d'interfaçage et d'autre part un fichier Java pour chaque classe de l'IDL possédant l'attribut `callback` (dans l'exemple le fichier `PointColoreStub.java`). Le schéma de classe final est exposé à la figure 3.11.

Le programme `test_p.ml` de la figure 3.12 montre l'utilisation des classes `Point` et `PointColore` à partir d'Objective Caml. L'opérateur de contrainte de type `:>` permet de contraindre le type d'un objet vers un surtype au sens de la relation de sous-typage.

Les trois premiers affichages appellent la méthode `display` de l'objet Java. Bien que la classe `point_colore_caml` redéfinisse aussi la méthode `getColor`, l'appel de `display` n'affiche pas l'information supplémentaire fournie par cette nouvelle méthode dans la mesure où l'appel à `display` reste dans le monde `Java`.

La classe `point_colore_mixte` permet de définir la méthode `getColor` en héritant de la classe abstraite `_point_coloreStub` d'Objective Caml qui encapsule la classe `PointColoreStub` de `JAVA`.

L'appel de la méthode `display` sur une instance de cette classe effectue les étapes de calcul décrites à la figure 3.14. La construction de la liste `l`, de type `point list`, montre le mélange de style fonctionnel et objet d'Objective Caml.

3.3.2 Implantation

Cette implantation utilise l'interface de bas niveau `camljava` [15] que l'on étend pour la communication de Java vers Objective Caml. La compilation d'un fichier *`.IDL`* construit les classes nécessaires pour une

[15] `pauillac.inria.fr/~xleroy/camljava`

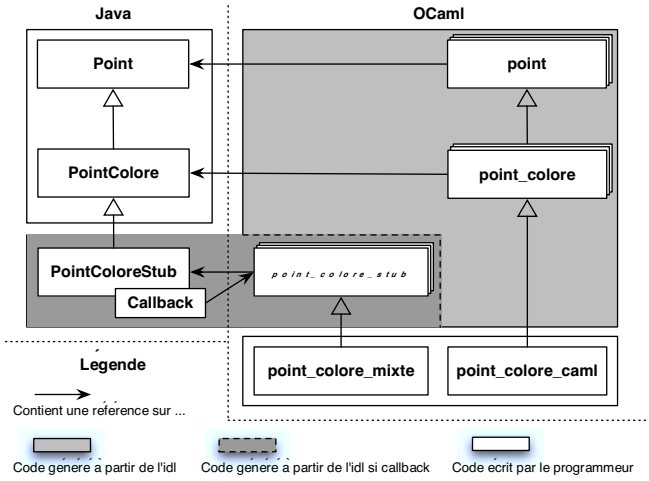


FIG. 3.11 – Relations entre classes

programme `test_p.ml`

```
open P;;

let p = new point 1 1;;
p#display();;                    (* 1 *)
let pc = new point_colore 1 3 "bleu";;
pc#display();;                   (* 2 *)

class point_colore_caml x y c  =
  object
    inherit point_colore x y c as super
    method getColor () =
      "[Caml"^(super#getColor ()) ^"]"
  end;;

let pcc = new point_colore_caml 1 3 "bleu";;
pcc#display();;                  (* 3 *)
```

```
class point_colore_mixte x y c =
  object
    inherit _point_coloreStub x y c as super
    method getColor () =
      "[Caml"^(super#getColor ()) ^"]"
  end;;

let pcm = new point_colore_mixte 1 3 "bleu";;
pcm#display();;                      (* 4 *)

let l = [(pcc :> point); (pcm :> point)];;
List.iter (fun x -> x#display()) l;; (* 5 *)
```

trace d'éxécution

```
(1,1)
(1,3):bleu
(1,3):bleu
```

```
(1,3):[Camlbleu]
(1,3):bleu(1,3):[Camlbleu]
```

FIG. 3.12 – Exécution d'un programme Java-O'Caml

communication sûre à travers cette bibliothèque de base.

`camljava`

L'interface `camljava` fait communiquer Objective Caml et Java à travers les interfaces avec C de chaque langage : Jni [74] pour Java et *external* [100] pour Objective Caml. Les méthodes Java (d'instance ou de classe) sont donc accessibles depuis Objective Caml en deux temps : recherche d'un identifiant de méthode par sa signature et appel de la méthode avec un tableau d'arguments sur un objet quelconque ; l'appel inverse suit le même schéma, les méthodes Objective Caml étant identifiées par leur nom. Cette séparation ne garantit pas, par exemple, le passage du bon nombre et du bon type des arguments, (et donc le maintien des piles dans un état cohérent lors du retour). `camljava` automatise le transfert des exceptions d'un monde à l'autre et assure le passage des arguments : type de base recopié, et référence sur les objets.

Du point de vue mémoire tout objet Java alloué en Objective Caml est une racine du GC de Java. Quand Objective Caml ne l'utilise plus, la racine Java est supprimée mais l'objet peut être conservé par le GC de Java s'il est référencé par un autre objet. Réciproquement les objets Objective Caml passés à Java sont eux aussi considérés comme des racines du GC d'Objective Caml le temps qu'ils résident en Java.

`camljava` a été étendu pour assurer que les communications entre les deux mondes s'effectuent toujours dans le *thread* principal. On s'aperçoit que `camljava` offre des possibilités de programmation riches mais dangereuses d'où la nécessité d'engendrer automatiquement les codes d'encapsulation et d'intégration au système de types.

Génération de code

On cherche à faire correspondre une classe Java décrite dans un fichier *IDL* à une classe Objective Caml. Comme toute instance de classe Java est considérée du type `Jni.obj` en Objective Caml, nous allons construire autour de ces objets Java de véritables objets Objective Caml. Pour faciliter la représentation de la hiérarchie de classes Java, nous introduisons une classe principale, appelée `top` en Objective Caml.

La figure 3.13 décrit les différents classes et types engendrés en Objective Caml et Java par la compilation d'une déclaration dans l'*IDL* selon la présence ou non de l'attribut **callback**.

classe	interface
- 1 type objet t - 1 classe encapsulante W de type t - 1 à n classes (C_i), sous-classe de W (1 par constructeur) - 1 fonction **instanceof** pour ce type - 1 fonction de *cast* pour ce type	- 1 type objet t - 1 classe encapsulante W de type t (1 par constructeur) - 1 fonction **instanceof** pour ce type - 1 fonction de *cast* pour ce type
avec **callback**	
- 1 classe souche (*stub*) - 1 à n classes abstraites (1 par constructeur) dont toutes les méthodes sont concrètes - 1 sous-classe Java	- 1 classe Java implantant l'interface - 1 classe abstraite dont toutes les méthodes sont abstraites

FIG. 3.13 – Compilation des déclarations

Dans l'exemple du `PointColore` défini à la figure 3.11 on définit un type objet `jPointColore` contenant toutes les méthodes décrites dans l'*IDL*

```
class type jPoint =
  object
    inherit JniHierarchy.top
    method _get_jni_jPoint : jni_jPoint
    method get_y : unit -> int
    method set_y : int -> unit
    method get_x : unit -> int
    method set_x : int -> unit
    method eq : jPoint -> bool
    method distance : unit -> float
    method display : unit -> unit
    method toString : unit -> string
    method rmoveto : int -> int -> unit
    method moveto : int -> int -> unit
  end
and jPointColore =
  object
    inherit jPoint
    method _get_jni_jPointColore : jni_jPointColore
    method setColor : string -> unit
    method getColor : unit -> string
  end
```

et deux classes «utilisateur» `point_colore` et `point_colore_default` héritant de la même classe capsule. Ces trois classes ont le type `jPointColore`.

```
class point_colore _p0 _p1 _p2 =
  let java_obj = _alloc_jPointColore () in
  let _ = _init_point_colore java_obj _p0 _p1 _p2 in
  (object (self) inherit _jPointColore java_obj end : jPointColore)
```

En présence de l'attribut **callback** une nouvelle classe dite «souche» est définie en Objective Caml pour encapsuler l'objet Java dont la classe a redéfini les méthodes de son ancêtre comme des appels vers la souche Objective Caml. L'utilisateur peut alors définir ses propres comportements en héritant d'une classe «utilisateur» (héritière directe de la classe souche). Le comportement par défaut de ces classes «utilisateur» est de propager l'appel vers la classe ancêtre Java.

La figure 3.14 qui illustre ce mécanisme les noms des classes de définition des méthodes sont indiquées en abrégé (par exemple *pc* pour `PointColore`).

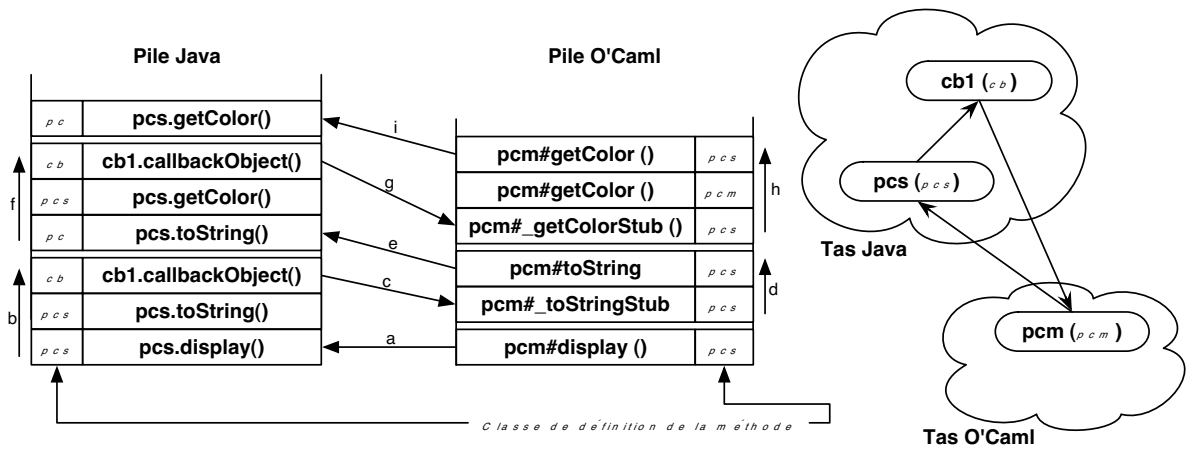


FIG. 3.14 – Détail d'un appel à display sur un `point_colore_mixte`

La création d'un objet de classe `point_colore_mixte` s'effectue en 4 étapes :
- allocation d'un objet Java `pcs` de classe `PointColoreStub`,
- allocation de l'objet Objective Caml
- initialisation de l'objet Objective Caml `pcm` avec la référence sur `pcs`
- initialisation de l'objet Java `pcs` qui se détaille :
  - appel à l'initialisateur de `PointColore`
  - allocation d'un objet Java `cb` de classe `Callback`
  - initialisation de l'objet `cb` avec une référence sur `pcm`

Certes la liaison tardive entre deux machines virtuelles provoquent quelques indirections supplémentaires. Pour cela les classes «utilisateur» correspondant à une souche **callback** sont déclarées abstraites et ne peuvent pas être utilisées par inadvertance sachant qu'il y a déjà une classe encapsulant l'objet Java.

Extension des modèles objets

`O'Jacaré` nous autorise à manipuler en partie les deux modèles objets. On illustre ces nouvelles possibilités par un héritage multiple en Objective Caml de classes Java et en effectuant une vérification dynamique de type (*downcast*) en Objective Caml.

Héritage multiple L'exemple suivant est repris de [41]. On définit deux hiérarchies de classes en Java : les objets graphiques et les objets géométriques. Chaque hiérarchie possède une classe `Rectangle`. Le programme Objective Caml suivant crée une classe héritant de ces deux classes Java.

fichier `q.idl`

```
package Graph;

[name rectangle_graph] class Rectangle
{ <init> (Point,Point);
  String toString();
}

package Geo;

[name rectangle_geo] class Rectangle
{
  <init> (Point Point);
  double compute_aera();
}
```

programme Objective Caml

```
open Q;

class rectangle_geo_graph p1 p2 =
object
  inherit rectangle_geo p1 p2 as super_geo
  inherit rectangle_graph p1 p2 as super_graph
end;;

let p1 = new point 10 10;;
let p2 = new point 20 20;;

let rgg = new rectangle_geo_graph p1 p2;;

Printf.printf "aera=%g
n" rgg#compute_area();;
Printf.printf "toString=%s
n" rgg#toString();;
```

Downcast Objective Caml n'autorise aucune opération de typage dynamique, néanmoins dans une interface avec Java cela est nécessaire au moins pour les objets provenant d'un calcul du côté Java. Dans l'exemple de la figure 3.12 on construisait une liste `l` de `point` or ceux-ci correspondaient à des points colorés. On autorise alors l'utilisation des fonctions de contraintes de type de `top` vers le type Objective Caml d'une classe Java. Ces fonctions déclenchent une exception si le type n'est pas compatible.

```
let l = [(pcc :> point); (pcm :> point)];;
let lc = Lisp.map (fun x -> top_to_jPointColore (x :> top)) l;;
```

On garde le côté explicite des contraintes de type propre à Objective Caml.

Modèles objet Le fait de vouloir enrichir le modèle objet de Java n'est pas récent. Odersky et Wadler introduisent du polymorphisme paramétrique à Java dans leur système [155]. Ils ajoutaient ainsi à Java un style de programmation proche d'Objective Caml. L'évolution de Pizza vers Generic Java [26] reprend ce nouveau style en introduisant un polymorphisme borné. Cette volonté de généricité se retrouve bien entendu du côté de C#. Le langage intermédiaire ILX [146] se veut plus général que MSIL pour faciliter les compilateurs de langages à la ML mais aussi pour introduire des *generics* en C#.

Du côté des langages fonctionnels la définition d'extensions objets comme Objective Caml autorise grâce à la laison tardive de nouvelles structurations logicielles[24]. Néanmoins le besoin d'une véritable hiérarchie de classes est réel et une ouverture du dogme du typage statique [33] devient nécessaire pour la réalisation de certains modèles de conception (*Design Patterns*).

3.4 Conclusion : évolution des interfaces

L'évolution des interfaces a été importante sur les vingt dernières années ; les langages fonctionnels ont fait partie du mouvement, quelquefois avec retard, mais le rattrapant au final. L'idée est toujours d'augmenter les fonctionnalités tout en assurant la sûreté. Il faut rappeler que cette notion d'interopérabilité est non seulement une mise en commun des réalisations dans différents langages d'implantation mais aussi un gage de liberté de choix de ceux-ci. Néanmoins il y a un risque inflationiste dans les *IDL*s, en particulier certaines nouvelles caractéristiques n'ont pas forcément du sens dans tous les langages. Par exemple le passage par copie des paramètres dans CORBA 2.0 a subi l'influence de *RMI* (*Remote Method Invocation*) de Java. On arrive déjà à envoyer à distance une instance des classes `Thread` qui peuvent alors redémarrer sur le receveur. À quand des continuations [123] passées par copie ? Il peut sembler raisonnable dans les cas de structures complexes de conserver un passage par référence.

Les difficultés d'exécution liées à la gestion mémoire et au multi-threading peuvent être simplifiées par l'utilisation d'un environnement d'exécution commun comme la machine virtuelle Java ou la plate-forme .NET. L'interfaçage de langages munis de *GC* utilise souvent des compteurs de références qui ne traitent pas les objets circulaires mixtes. Le multi-threading multi-langages lui aussi pose des difficultés par la création de *threads* dépendant d'au moins deux ordonnanceurs. Pour traiter ces problèmes nous allons adapter `O'Jacaré` pour le compilateur O'CAMiL décrit au chapitre 1.

L'interopérabilité entre langages peut gagner aussi du côté dynamique de l'interface et du chargement de code. L'embarquement du compilateur d'un langage (ce qui est le cas pour le *toplevel*) permet de compiler à la volée des programmes issus de l'exécution de l'application. On peut alors prévoir d'embarquer les programmes de génération de code d'un *IDL*. Cela autorise un interfaçage dynamique à la volée. La vérification de types peut suivre la technique utilisée pour `O'Jacaré` qui consiste, suite au typage statique de la partie CAML, à vérifier à la fin d'une déclaration d'une classe l'existence de la classe Java correspondante. Cette phase d'élaboration revient à une édition de liens retardée. On obtient ainsi une interopérabilité dynamique sûre en évitant un interfaçage trop lourd des hiérarchies de classes Java ou C#.

Chapitre 4

Environnements de développement

Le chapitre 1 a été en grande partie consacré à la portabilité des applications ou des bibliothèques développées en ML. Si les applications Objective Caml sont multi-plateformes, on imagine aisément qu'il y aura des programmeurs sur ces différentes plates-formes. Pour cela nous devons penser à un environnement de développement portable. On entend par environnement de développement un ensemble d'outils facilitant grandement la tâche du programmeur. On peut lister les outils suivants : éditeur structuré, trace de l'exécution, exploration de la pile et du tas, mesure de performances, aide à la construction d'applications, production des documentations, gestion de versions, etc. Ces outils peuvent prendre la forme d'un ensemble de commandes séparées (à la Unix) ou être intégrés dans un *IDE*[1]. L'existence et la qualité d'un environnement est un des éléments de choix d'un langage pour de nombreux programmeurs.

On peut voir Emacs[2] comme un tel outil car il intègre le lien entre l'éditeur structuré et la compilation. Sur les plates-formes Unix/X-window, Emacs est probablement l'outil le plus utilisé pour Objective Caml. Néanmoins sa prise en main peut rebuter le programmeur Windows.

Les outils de mise au point des langages ML ou plus généralement des langages statiquement typés polymorphes se heurtent aux schémas de compilation classiques de ces langages qui profitent du fait qu'un programme soit bien typé pour ne pas conserver d'information de typage à l'exécution. Il sera alors difficile d'explorer les valeurs (et de les modifier) à l'exécution à l'aide d'un outil de mise au point. Nous avons déjà évoqué cette difficulté au niveau de la compilation vers une machine abstraite typée (voir chapitre 1) et au niveau de la persistance (voir chapitre 2). Une décennie après l'article «Runtime Tags Aren't Necessary» d'Andrew Appel [8], l'argument de pure efficacité n'est plus de mise et l'ouverture au monde extérieur ainsi que les possibilités d'introspection priment. L'argumentation dépasse en fait la mise au point des programmes pour laquelle il est indispensable de construire ou reconstruire les types des valeurs manipulées à l'exécution.

Les travaux présentés dans ce chapitre sont moins aboutis que dans les précédents. Il faut les considérer comme des pistes que nous suivrons dans un futur proche. La première section rappelle brièvement les outils standards utilisés en Objective Caml. La deuxième section décrit un *IDE* portable pour Objective Caml, appelé `ocaide`[56][3], qui contient les premières briques pour un outil plus complet. Il est bâti sur l'outil de construction d'outils Eclipse[4] et est implanté en Java. Ce travail est principalement dû à Alexandre Deckner et Léonard Dallot encadrés par mes soins. La troisième section présente un travail commun avec Pascal Manoury et Bruno Pagano sur une reconstruction de types de coût raisonnable. Cette reconstruction retardée est proposée à partir d'une information partielle de type incluse dans les valeurs créées à l'exécution.

[1] *Integrated Development Environment.*
[2] `www.gnu.org/software/emacs/emacs.html`
[3] `glalex2.free.fr/eclipse/odtupdate`
[4] `www.eclipse.org`

4.1 Environnement standard pour Objective Caml

L'environnement de développement standard de la distribution Objective Caml repose sur un ensemble d'outils séparés fonctionnant principalement sous Unix. Emacs peut être considéré comme l'*IDE* d'Objective Caml. On trouve plusieurs modes de configuration Emacs pour ce langage. Il est donc possible de compiler un module ou une application, mais aussi de lancer le *toplevel* dans une fenêtre de l'éditeur permettant d'évaluer des parties de programme pour des tests immédiats. En conjonction avec des outils externes comme `ocamldep` et `ocamlmake` la génération des dépendances et des makefiles est facilitée. Le gestionnaire de version CVS étant lui-aussi appelable dans Emacs. La génération de documentations utilise la commande `ocamldoc` intégrée depuis peu dans la distribution du langage. On retrouve par ces différentes commandes de nombreuses fonctionnalités d'un véritable *IDE*.

De plus les outils de *debug* et de *profiling* d'une application peuvent être lancés pour affiner la mise au point. Le *debugger* d'Objective Caml, dû à Jérôme Vouillon, permet d'explorer la pile d'exécution et possède la possibilité de rejouer une exécution. Par contre l'exploration des valeurs et leurs modifications se heurte à la connaissance à l'exécution du type des valeurs. Le *profiler* du compilateur natif d'Objective Caml produit un résultat à la manière de la commande `gprof`.

L'ensemble de ces outils est réalisé pour le monde Unix. Ils ne s'adaptent pas bien à la plate-forme Windows. Par exemple le *debugger* utilise l'appel système `fork` pour dupliquer les processus d'exécution et pour pourvoir rejouer une session. Il est certes possible d'installer un environnement «à la Unix» sous Windows comme Cygwin[5], mais même ainsi on ne retrouve pas le confort attendu d'Unix. Pour cela on présente les premières briques d'un *IDE* portable.

4.2 *IDE* portables : Eclipse

Eclipse est développé par un *consortium*[6] d'industriels qui fédère une communauté importante d'utilisateurs, développeurs et chercheurs. Eclipse est défini de la manière suivante :

> «*Eclipse is a kind of universal platform - an open extensible IDE for anything and nothing in particular.*»

C'est en même temps une plate-forme de développement d'*IDE* spécifiques et un environnement d'exécution de ces *IDE* pour la construction d'applications.

Eclipse offre un mécanisme de *plugins* pour l'exécution des nouveaux *IDE* ou outils spécifiques. Au démarrage d'Eclipse les différents *plugins* détectés sont accessibles à l'utilisateur. Pour faciliter l'écriture de *plugins*, Eclipse offre de nombreuses bibliothèques Java portables pour la manipulation d'un éditeur structuré, la liaison avec des commandes externes, la construction d'interfaces utilisateur, etc.

On trouve de nombreuses présentations, documentations et articles sur Eclipse. On peut citer la synthèse [114] [7] et la présentation de l'écriture du premier *plugin* [106][8]

4.2.1 Architecture d'Eclipse

Eclipse est structuré en sous-systèmes (implantés eux-mêmes par des *plugins*) qui font appel au *Platform Runtime* (environnement d'exécution de la plate-forme). Le *Workspace* (environnement de travail) fournit une *API*[9] (interface de programmation) pour la gestion des ressources liées au système de fichiers. Le *Workbench* (plan de travail) correspond à l'interface utilisateur. Son *API* définit les points d'extension pour les éditeurs, menus ou vues. Il utilise JFace basée sur SWT[10] pour la construction des interfaces utilisateurs. Le *Help*

[5] `www.cygwin.com`
[6] `www.eclipse.org/org/index.html`
[7] `www.eclipse.org/whitepapers/eclipse-overview.pdf`
[8] `www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html`
[9] *Application Programming Interface*
[10] *Standard Widget Toolkit.*

System (système d'aide) est à utiliser pour créer les aides en format HTML, construire les tables de matières et la recherche par mot clé. Le *Team Support* permet une gestion de versions multi-utilisateurs en supportant CVS[11].

4.2.2 *Plugin* pour Objective Caml

Tous les sous-systèmes sont implantés en Java. Le *plugin* `ocaide` pour Objective Caml est réalisé en Java quitte à appeler plusieurs outils propres à la distribution Objective Caml en récupérant les résultats pour les passer aux différents composants de l'espace de travail.

`ocaide` définit ce qu'est un projet Objective Caml, permet de créer des configurations de compilation spécifiques et expose le projet dans une perspective CAML.Celle-ci découpe l'espace de travail en quatre zones principales (voir figure 4.1) :
- la zone de navigation dans le projet («Navigator») ;
- la zone d'édition ;
- la zone de navigation dans un fichier («Outline») ;
- la zone d'informations de la compilation («Tasks») et de l'exécution «Console»).

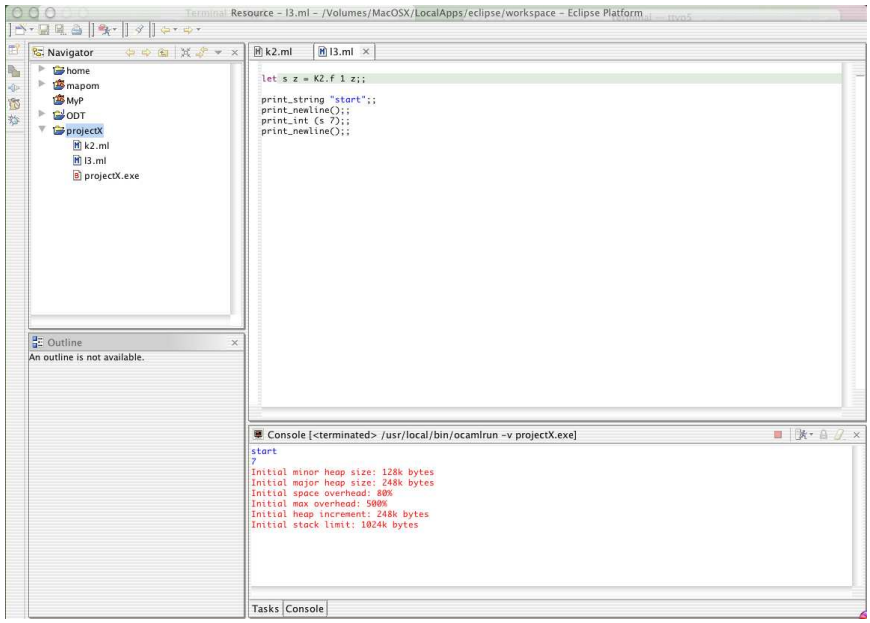


FIG. 4.1 – Capture d'une session Eclipse (MacOSX)

Les informations issues de la compilation du projet sont répercutées dans l'éditeur (*warnings* et erreurs).

Une partie du code écrit consiste à récupérer des informations produites par les outils de la distribution Objective Caml. Les dépendances des différents modules d'un projet sont calculées par la commande `ocamldep`. Les *warnings* et erreurs de compilation sont le fruit du compilateur de code-octet `ocamlc`. La navigation dans les déclarations globales d'un module utilise l'option `-dparsetree` du compilateur qui produit l'arbre de syntaxe d'un module. Les analyseurs lexicaux et syntaxiques sont compilés par les commandes `ocamllex` et `ocamlyacc`. Toutes ces commandes sont précisées lors de la création d'une configuration de projet. L'exécution d'un projet lance la machine virtuelle Objective Caml sur un exécutable de code-octet. Pour conserver sa caractéristique de portabilité, `ocaide` utilise le compilateur code-octet.

[11] *Concurrent Versions System.*

La deuxième partie importante du développement consiste à paramétrer l'éditeur structuré pour la syntaxe Objective Caml et l'utilisation des bibliothèques du *Workbench* pour la gestion du projet. Cela inclut l'interface d'utilisation du gestionnaire de versions (*CVS*).

L'installation d'`ocaide` peut être effectuée *via* une archive ou à distance à partir d'un site *web*. Les mises à jour n'en sont que plus aisées.

Exemples d'utilisation

La première session (figure 4.1) montre les quatre fenêtres principales de l'espace de travail de la perspective Caml[12].

La fenêtre «Navigator» permet de sélectionner un projet et ses ressources (dans l'exemple le projet `projetX` et les fichiers `K2.ml` et `I3.ml`). La fenêtre de l'éditeur affiche le fichier `I3.ml` correspondant au point d'entrée du projet. La compilation du projet engendre un exécutable `projetX.exe` qui est lancé dans la fenêtre «Console».

À chaque sauvegarde d'un élément du projet, il est recompilé ainsi que les ressources dépendant de lui. Vu la vitesse de compilation, l'utilisateur peut avoir la sensation que le *plugin* lui-même effectue le typage. Il n'en est rien, il y a bien un appel au compilateur Objective Caml.

Le projet `App1` des figures 4.2 et 4.3 contient les fichiers `main.ml`, `mod1.ml` et a importé le module `Queue` de la distribution standard.

Lors de l'édition d'un fichier, la fenêtre «Outline» permet une navigation dans les déclarations globales comme par exemple la fonction `copy` de `queue.ml` à la figure 4.2. La fenêtre «Tasks» indique les *warnings* et la première erreur de compilation pour chaque fichier qui sont reportées dans l'éditeur. Cette navigation dans

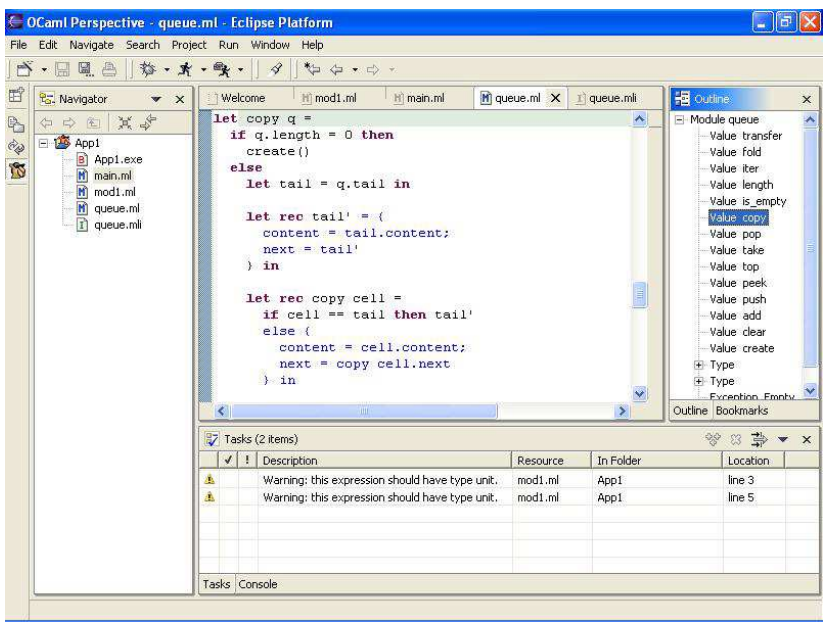


FIG. 4.2 – Capture d'une session Eclipse (Windows)

[12] visible dans les icones du bord gauche de la fenêtre principale.

les sources s'effectue en utilisant l'information transmise par la compilation (`-dparsetree`). Les expressions globales sont nommées `Eval` (voir figure 4.3).

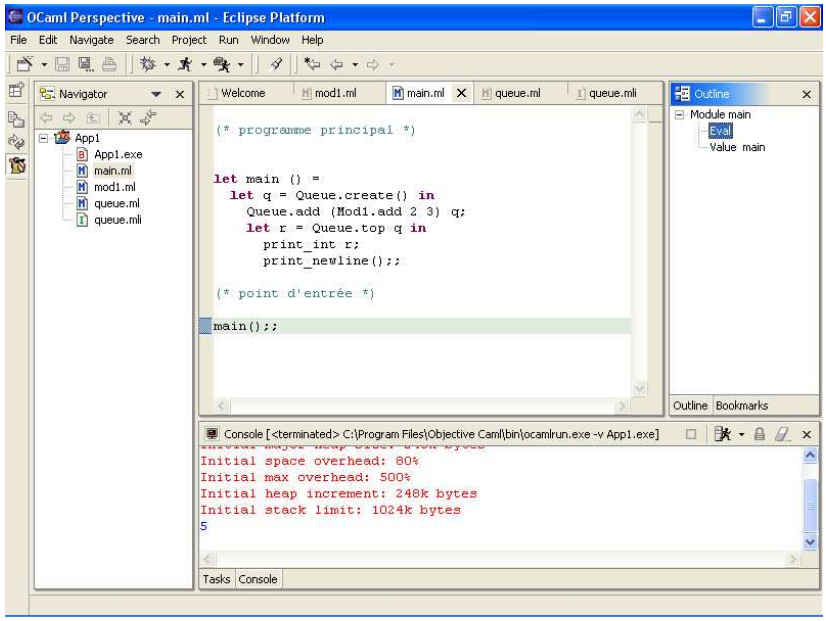


FIG. 4.3 – Capture d'une session Eclipse (Windows)

4.2.3 Remarques

Cette présentation succincte de l'outil `ocaide` a montré un début d'environnement intégré portable. La perspective Caml réalise la liaison entre informations des outils de la distribution Objective Caml. Le *debugger* d'Objective Caml fonctionnant uniquement sous Unix n'a pas pu être intégré pour conserver la portabilité.

L'*IDE* `ocaide` est écrit en Java. Une partie de son code sert à interfacer des outils d'Objective Caml avec Java. La prochaine version utilisera l'*IDL* `O'Jacaré`, décrit au chapitre 3, pour pouvoir développer en Objective Caml et faciliter l'interaction des outils propres au compilateur avec les bibliothèques d'Eclipse.

4.3 Types à l'exécution

Les langages polymorphes paramétriques, statiquement typés à la ML, n'ont pas besoin de conserver d'informations de type sur les valeurs construites pour une exécution sûre, *i.e.* sans erreur d'exécution due aux types.

Et pourtant il existe de nombreux cas où il est nécessaire de manipuler une information de types à l'exécution. On peut citer les domaines suivants :

- pour l'implantation de GC sans marque (*tag*) ;
- pour les fonctions polymorphes dépendant du type des arguments comme `print` ou `eval` ;
- pour la persistance des valeurs (`write` et `read`) ;
- pour la communication entre processus ;
- pour les outils de mise au point comme l'exploration de valeurs et leur modification ;
- pour des extensions du langage pour la surcharge, le sous-typage et le *cast*.

Nous avons déjà rencontré certains de ces problèmes dans les chapitres précédents, mais ils étaient traités de manière spécifique. La manipulation de types à l'exécution est un cadre plus général, bien qu'il n'y ait

pas encore de modèle général.

4.3.1 Modèles et techniques d'implantation

On peut essayer de distinguer ces différents usages du point de vue du langage et de son implantation. Tout d'abord les valeurs typées dynamiquement sont-elles utilisées explicitement ou implicitement ? Dans le premier cas le langage intègre un «constructeur» (`dynamic`) et la manipulation du type associée à une telle valeur s'effectue par un filtrage sur ce type (`type-case`). Son utilisation est donc limitée aux valeurs ainsi construites explicitement par le programmeur [101, 1].

Une autre vision est de dire que toutes les valeurs sont des `dynamics`. Pour éviter une écriture alourdie, chaque valeur possède un type qui sans indication explicite du programmeur doit être accessible. On distingue alors le cas d'une exécution spéciale du programme, comme par exemple pour un *debugger*, de l'exécution habituelle. Ce dernier cas englobe les besoins de gestion mémoire, de la persistence des valeur et de la communication. Autant un outil de mise au point peut avoir des performances faibles, autant l'introduction du typage dynamique ne doit pas être coûteux dans le cas d'une exécution normale. Les difficultés rencontrées sont liées aux performances désirées.

On distingue alors trois cas :

1. toutes les valeurs ont un type dynamique associé ;
2. le type dynamique de chaque valeur peut être reconstruit ;
3. le type dynamique d'une valeur peut être reconstruit partiellement.

La construction systématique d'un type associé immédiatement à chaque valeur construite nécessite de rejouer une partie de l'algorithme de typage de la phase de compilation. À chaque construction de valeur (application, constructeur), son type est calculé immédiatement par unification. L'environnement d'exécution contient l'environnement de typage pour les déclarations. C'est la construction de types la plus précise et la plus lente. L'histoire du calcul est toujours à jour. Les autres algorithmes vont se comparer avec tant pour la précision de la reconstruction que pour l'efficacité d'exécution.

La première variante est de retarder le calcul du type dynamique associé à une valeur. Celui-ci est alors calculé à la demande. Dans le cadre du polymorphisme paramétrique la seule histoire du calcul (par exploration de la pile) ne suffit pas. Il existe trois cas où celle-ci a disparu : l'application partielle construit bien une fermeture mais ne conserve pas le type de l'argument, l'affectation d'une référence change sa valeur sans toucher son type, le déclenchement d'une exception efface une partie de l'histoire. Pour cela des informations de la phase de typage doivent être ajoutées.

Aditya dans la construction d'un *debugger* pour Id [3] ajoute une information sur les paramètres de types pour certaines fonctions polymorphes ne pouvant pas reconstruire précisément les types des valeurs de l'environnement d'une fermeture. Il détermine les sites d'appel de fonctions où l'information de types peut être perdues sans passage explicite. L'algorithme de reconstruction de types utilise alors l'information contenue dans la pile et l'information supplémentaire ainsi propagée. Cette technique est aussi utilisée pour un *debugger* SML [150].

Pour les GC une implantation sans marque peut utiliser un GC à racines ambiguës [158, 89] et tenter de discriminer les valeurs. Cette technique est utilisée dans les GC pour C [22] comme dans CeML[31]. Une autre voie a été explorée dans le but de reconstruire le type des valeurs quand cela est nécessaire. Dans [72] les types sont seulement partiellement reconstruits à partir de la pile et de fonctions de reconstruction spécialisées pour certaines fonctions polymorphes. La manière dont les valeurs de l'environnement d'une fermeture ont été construites n'est plus accessible d'où la nécessité de conserver une partie de cette information. La reconstruction reste partielle, néanmoins dans ce cas l'utilisation de l'information construite reste correcte pour le GC comme l'indique Benjamin Goldberg :

> «*any object whose type cannot be reconstructed by the GC is garbage.*»

Aditya dans [4] reprend sa technique pour le *debug* en ajoutant une information de type comme arguments supplémentaires (qui se trouveront dans la pile) à certaines fonctions polymorphes. Andrew Tolmach dans [151] ajoute systématiquement une information de types pour toutes les fonctions polymorphes.

4.3.2 Utilisation du tas

D'un point de vue implantation, une partie de l'information de types provient de l'information du typage de la phase de compilation qui dans certains cas doit être affinée au moment de l'exécution. Cette information peut être contenue soit dans la pile, soit dans le tas. De plus, cette information de type statique, connue à la compilation, doit être traitée pour reconstruire le type dynamique exact des valeurs. Enfin le coût *mémoire* induit par ce stockage supplémentaire et le coût *temps* provoqué par la reconstruction de type doivent être maitrisés si l'on désire s'en servir pour de nouveaux traits du langage. En particulier cette reconstruction de types gagne à être paresseuse pour éviter un surcoût trop important si l'on ne s'en sert pas.

Pour cela on cherche à stocker de manière légère une information de type associée aux valeurs ; soit dans les valeurs, soit à l'extérieur de celles-ci dans le tas. Cette information supplémentaire a des conséquences pour les algorithmes de GC (Mark & Sweep ou Stop & Copy). Pour minimiser l'occupation mémoire on présente un nouvel algorithme de Stop & Copy. Celui-ci découpe le tas en trois zones : une zone pour les valeurs, une zone pour le déplacement de celles-ci et une zone pour conserver l'information de type. Cette version est ensuite optimisée pour confondre la zone de déplacement et la zone d'informations de type.

Informations de type dans les valeurs

Notre but est d'associer un «type» pour chaque valeur allouée. Cette information de type doit être directement accessible à partir du pointeur de la valeur allouée. Une première possibilité est d'étendre la valeur par ce type. Le surcoût mémoire est d'au moins un mot mémoire pour respecter l'alignement. Pour un GC Mark & Sweep cela peut être acceptable, mais pour un Stop & Copy cela est plus cher dans la mesure où cet espace additionnel doit lui aussi être dupliqué dans l'espace de déplacement.

Une autre voie est de conserver cette information de type à l'extérieur de la valeur associée. Pour optimiser l'occupation mémoire on peut utiliser une partie séparée du tas comme un tableau de cellules pouvant être de taille inférieure à un mot mémoire. Pour garder l'association entre valeur et type, les types sont rangés dans un espace homotétique. Cela devient plus raisonnable pour un Mark & Sweep, mais il y a toujours ce double coût (pour la partie type) pour un Stop & Copy.

Notre idée est alors d'utiliser un nouvel algorithme de Stop & Copy avec seulement un espace pour les types et utilisant une homothétie entre espace de valeurs et espace de types. On appelle «zone mirroir» l'espace de types.

Stop & Copy en place

L'algorithme Stop & Copy que nous présentons nécessite un alignement des valeurs sur 64 bits, par conséquence tous les objets alloués auront un nombre pair de mots de 32 bits. On obtient la même allocation dans le tas pour les machines à mots mémoire de 32 ou de 64 bits.

Cet algorithme permute les valeurs de l'espace mémoire (*value-space*) en utilisant l'espace (*displacement-space*) tout en conservant les informations de l'espace (*mirror-space*). L'information de typage pour chaque cellule de *value-space* est conservé dans *mirror-space* au même emplacement.

La figure 4.4 représente la double allocation d'une cellule. Les v_i représentent la valeur d'une cellule, t_1

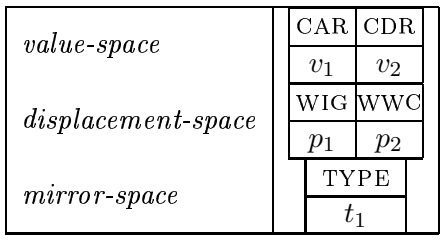


FIG. 4.4 – Espaces des valeurs, de déplacement et mirroir

son information de type associée et les p_i le déplacement des cellules effectué par le GC. Le mot mémoire WIG (*Where I Go*) est utilisé pour contenir la future adresse de la cellule et le mot mémoire WWC (*Which Will Come*) recevra l'adresse de la cellule déplacée.

Cet algorithme est découpé en quatre étapes. Les deux premières étapes déterminent quelles cellules vont bouger et où elles vont aller. Cette information est conservée dans l'espace *displacement-space*. La troisième étape met à jour les pointeurs contenus dans les cellules en fonction de leur futur déplacement. Ces nouvelles adresses seront correctes après le déplacement. La dernière étape effectue le déplacement des cellules et de leur information de types. Il y a trois cas de déplacement : déplacement simple d'une cellule vers un emplacement libre, déplacement d'une structure linéaire chaînée ou déplacement d'une structure circulaire.

L'algorithme est présenté dans sa version optimisée, où les espaces *mirror-space* et *displacement-space* sont confondus comme à la figure 4.5. Dans notre approche de reconstruction de types dynamique il est possible

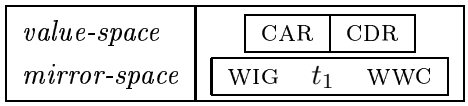


FIG. 4.5 – Rassemblement de l'espace de déplacement et de l'espace des types

de coder l'information de types (types de surface) avec l'information de déplacement d'un pointeur dans une même cellule (de deux mots) de l'espace *mirror-space*.

4.3.3 Types sans espace de type

Les champs WIG and WWC et l'information de type sont utilisés pour la réalisation du GC. L'idée principale de ce GC est de calculer une permutation de cellules de *value-space* vers *value-space* et de le sauver dans les champs WIG et WWC.

On présente cet algorithme dans un *pseudo-C* en le détaillant sur un petit exemple. On construit des valeurs ML par le code suivant :

```
let l  =  [‘b‘;‘c‘;‘d‘] ;;
let a  =  match ( [‘a‘] , l )
          with  p  ->  ( fst p , snd p ) ;;
```

Tout d'abord ce code alloue deux listes dans le tas, puis construit une cellule pour la paire contenant ces deux listes. Le résultat du calcul de l'expression `a` est une autre paire. Celle-ci est considérée comme une racine, alors que la première n'est plus utile et peut être récupérée. En utilisant une implantation particulière des listes (autorisée grâce au typage dynamique), le tas contient les structures décrites à la figure 4.6. Le tas lui-même est représenté à la figure 4.7. On peut noter que l'espace *mirror-space* ne contient que des types et les p_i valent toutes le pointeur nul. Cela doit rester vrai quand le GC termine.

À chaque étape on indique son pseudo-code et on montre l'état du tas. On suppose qu'il y a une discrimination sûre entre valeurs immédiates et pointeurs (soit codée dans la valeur, soit dans le type correspondant).

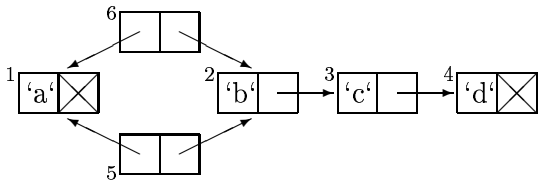


FIG. 4.6 – Une valeur Caml

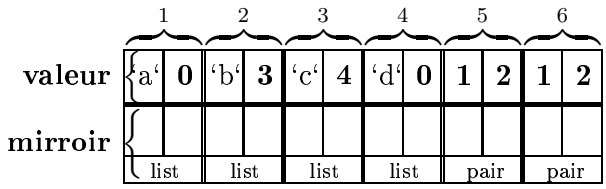


FIG. 4.7 – État du tas avant GC

Soient les variables `VS` et `MS` correspondant aux espaces *value-space* et *mirror-space*. On note `VS[wh]`, la wh-ième cellule, `CAR[wh]` la partie `CAR` de la wh-ième cellule, `CDR[wh]` la partie `CDR` de la wh-ième cellule, `WIG[wh]` est le pointeur p_1 et `WWC[wh]` le pointeur p_2 de la wh-ième cellule.

La variable `free` correspond à un pointeur sur la prochaine cellule libre. Comme on suppose que chaque valeur est marquée pour discriminer une valeur immédiate d'un pointeur, on introduit les deux fonctions suivantes :

```
int is_addr_car (addr);
int is_addr_cdr (addr);
```

Les deux premières étapes calculent la permutation des cellules. La future position de chacune est sauvée dans le champs `WIG` correspondant, et l'adresse de la cellule à venir dans le champs `WWC`. Cela est réalisé par la fonction suivante :

```
void Treat(addr a) {
  if ( WIG[a] == NULL) {
    WIG[a] = free;  WWC[free] = a;
    free++;
  }
}
```

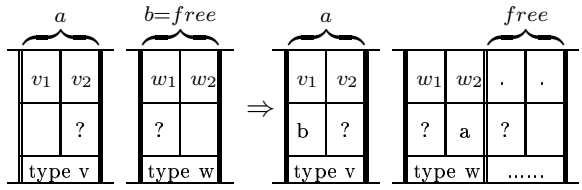


FIG. 4.8 – Appel de Treat(a)

Étape 1) Traitement de l'ensemble des racines (`RS`) :

```
free=1;
for (i=1;i<length(RS);i++) Treat(RS[i]);
```

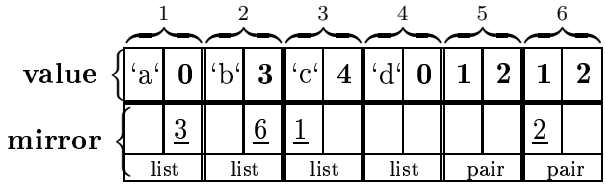


FIG. 4.9 – Tas après l'étape 1

Étape 2) Traitement de `WIG` et `WWC` pour chaque cellule déplacée, y compris celles déplacées à l'étape 2 :

```
for ( wh=1 ; wh < free ; wh++ ) {
  addr a = WWC[wh];
  if (is_addr_car(a))  Treat(CAR[a]);
  if (is_addr_cdr(a))  Treat(CDR[a]);
}
```

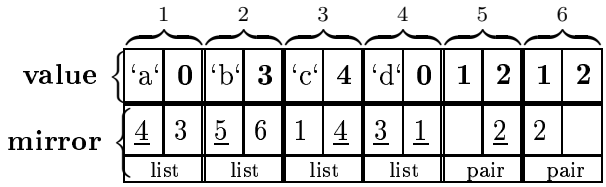


FIG. 4.10 – Tas après l'étape 2

Étape 3) Mise à jour des pointeurs :

```
 for (i=1;i<length(RS);i++) RS[i]=WIG[RS[i]];
 for ( wh=1 ; wh < free ; wh++ ) {
   addr a = WWC[wh];
   if (is_addr_car(a))  CAR[a] = WIG[CAR[a]]
   if (is_addr_cdr(a))  CDR[a] = WIG[CDR[a]]
 }
```

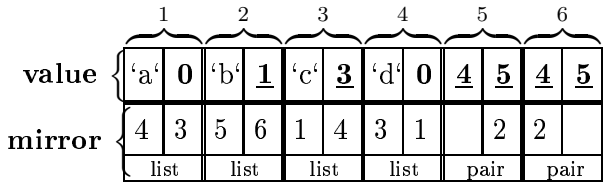


FIG. 4.11 – Tas après l'étape 3

Étape 4) Il est temps d'effectuer les déplacements ; pour cela on a besoin de deux fonctions auxiliaires pour déplacer une cellule et un ensemble de cellules chaînées.

```
void move (addr src, addr dest) {
  VS[dest] = VS[src];
  MS[dest]->type = MS[src]->type;
  WWC[dest] = NULL;
  WIG[src] = NULL;
}

void move_chain (addr a) {
  while (a != NULL) {
    addr b = WWC[a];
    move(b,a);
    a = b;
  }
}
```

Alors l'algorithme utilisé pour déplacer les cellules est le suivant :

```
for ( wh=1 ; wh < free ; wh++ ) {
  addr a = WWC[wh];
  if (a != NULL)  {
    /* a reste en place */
    if (WIG[wh] == a ) {
       WIG[wh] = NULL;
       WWC[wh] = NULL;
    }
    else
      /* mouvement simple */
      if (WIG[wh]==NULL) move(a,wh);
      else {
        addr b = WIG[a];
        while ( b!=a && WIG[b]!=NULL)
          b=WIG[b];
        /* mouvement d'une liste chaînée */
        if (WIG[b]==NULL)
          move_chain(WIG[b])
        /* mouvement circulaire */
        else {
          move(a,buffer);
          move_chain(a);
          move(buffer,WIG[a]);
        }
      }
  }
}
```

Pour les déplacements de structures circulaires, ce traitement est suffisant. Les seuls cycles possibles sont ceux concernant la première cellule (la variable `a` dans ls pseudo-code). Si c'était le cas, une cellule pourrait avoir deux destinations. Et cela ne peut pas arriver. Dans notre exemple, on détecte un cycle pendant le traitement de la première cellule : $1 \leftarrow 3 \leftarrow 4 \leftarrow 1$, aussi la variable `buffer` est utilisée pour sauver le contenu de la cellule 3, et les cellules sont déplacées. Après ce mouvement, on traite la cellule 2 qui peut alors bouger vers la cellule libre 5. À la fin, la cellule 6 est bougée vers la cellule 2. Le résultat correspond aux figures 4.12 et 4.13.

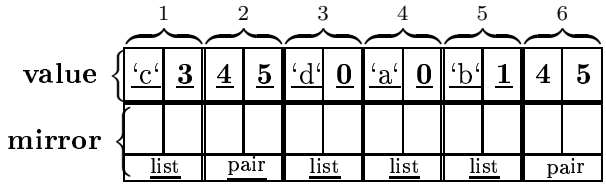


FIG. 4.12 – Tas après l'étape 4

Dans le but de simplifier la présentation, nous n'avons utilisé uniquement des objets de taille deux. Pour entrer dans le cas général, il est nécessaire d'itérer les deux premières étapes pour toutes les cellules jointes d'une même valeur. On peut marquer ces cellules jointes en réservant un bit d'information de type pour indiquer que la cellule courante est une partie d'une valeur dont la cellule suivante fait partie. Ainsi quand deux cellules sont jointes, on garantit que leurs nouvelles positions seront déterminées successivement et qu'elles resteront contiguës. On peut remarquer que l'étape de déplacement est complètement indépendante de la nature des valeurs déplacées. Les deux premières étapes ne modifient pas les données. L'étape trois peut être inversée à tout moment. Et avant que l'étape quatre ne démarre, il est possible d'interrompre le GC.

Cet algorithme n'autorise pas de références à l'intérieur d'une donnée. Pour autoriser de tels pointeurs, il

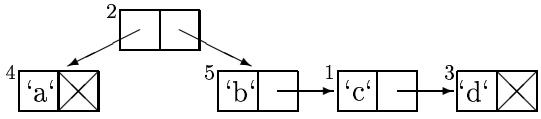


FIG. 4.13 – Valeur Caml survivante

faut modifier les deux premières passes.

Performances

Avant d'aller plus loin, on cherche à évaluer les risques de perte de performance de notre algorithme. La figure 4.14 compare celui-ci (colonne «en place») avec un Stop & Copy classique (colonne SC) sur un programme de manipulation de listes : construction de la liste des nombres premiers en utilisant le crible d'Erathostene. Ces temps sont donnés sur un SPARC 10. Il y a deux versions de l'algorithme : Era1 et Era2 qui optimise l'ensemble des racines, aussi le tas est plus petit après GC.

Era 1			
heap size	GC nb	SC	en place
10000	830	18.5 s	24.8 s
100000	66	17.1 s	19.9 s
1000000	6	18.0 s	20.4 s
Era 2			
10000	80	9.4 s	9.7 s
100000	7	9.5 s	9.8 s
1000000	0	10.0 s	10.9 s

FIG. 4.14 – Tests de base

Les temps correspondent au calcul complet qui inclut le temps du GC. Le Stop & Copy classique ne manipule pas les types et est légèrement plus rapide. Le pire des cas arrive à la première ligne (figure 4.14). La petite taille du tas entraîne un nombre important de GC, le prix à payer est de devoir déplacer plus souvent les cellules survivantes. Dans ce cas, l'algorithme «en place» est environ sensiblement plus lent que le Stop & Copy classique.

L'augmentation mémoire est raisonnable. La taille de la pile reste la même. La taille du tas grossit seulement pour les structures d'un nombre impair de cellules. Dans le pire des cas, si toutes les structures ont besoin de trois mots mémoire, alors la perte s'élève à 25% du tas.

Ce cas est rare dans la mesure où un programme ML alloue des valeurs de taille différente comme les n-uplets ou les fermetures. Ce type de GC permet d'optimiser la représentation des données dans le tas. Par exemple un nombre à virgule flottante (`double`) peut être stocké sur deux mots sans marque dans sa valeur. De plus l'alignement 64 bits est fréquemment utilisé et dans ce cas il n'y a pas ce perte d'espace.

Il semble possible d'utiliser cet algorithme pour des GC à générations ou pour des GC mixant des algorithmes Mark & Sweep et Stop & Copy. L'espace de types peut être utilisé pour sauver les numéros de générations des cellules qui survive à un GC.

4.3.4 Reconstruction de types

Bien qu'ayant une information de type pour chaque valeur, le problème est maintenant de reconstruire leur type durant l'exécution. Pour cela on propose différentes reconstructions de type en fonction de l'usage qui en

sera fait. On présente tout d'abord la reconstruction de types pour les valeurs immédiates pour s'intéresser ensuite aux valeurs non fonctionnelles puis aux valeurs fonctionnelles.

Valeurs immédiates

Les valeurs immédiates ne sont pas allouées dans le tas, néanmoins elles peuvent apparaître dans des structures allouées. On cherche à garder la représentation uniforme des données sans ajouter d'informations dans la pile.

Pour distinguer les pointeurs des valeurs immédiates pendant le GC, on adopte le solution usuelle d'un bit de marque. Mais l'on désire aussi distinguer les types des valeurs immédiates. Par exemple en ML, les entiers, les caractères et les constructeurs constants sont codés de la même manière. Pour cela il est nécessaire d'utiliser plus d'espace pour encoder le type d'une valeur immédiate. Pour simplifier ce problème, on ne conserve que les entiers comme valeurs immédiates ; toutes les autres valeurs immédiates seront allouées et leur type seront inscrits dans l'espace mirroir. Ces valeurs peuvent être allouées une fois pour toutes si le nombre de valeurs possibles d'un type est petit comme pour les constructeurs ou les caractères.

Au final, on obtient (facilement) les types des valeurs immédiates (*i.e.* les entiers) sans utiliser une information dans la pile.

Valeurs allouées non fonctionnelles

Pour les types monomorphes (sans paramètre de type), l'information du type global est son type. Mais pour les déclarations de types paramétrés, il est nécessaire de retrouver ses paramètres de type.

Pour chaque type on définit une fonction générique `type_of_val` qui calcule le type à l'exécution d'une valeur à partir de son type global. Le cas monomorphe est immédiat. Pour chaque type paramétré, on définit à la compilation une fonction spécialisée de reconstruction pour ce type.

Par exemple, si on déclare le type suivant :

déclaration de type

```
type ('a,'b) k = Z of 'a | U of 'b * 'a;;
```

alors on définit la fonction de reconstruction associée :

fonction de reconstruction

```
let build_type_k x = match x with
  Z u     -> (type_of_val u, undef) type_k
| U (u,v) -> (type_of_val u, type_of_val v)
               type_k ;;
```

où `type_k` est le type global de `k`, et `undef` indique un paramètre de type non déterminé. La fonction principale `type_of_val` retourne le type complet d'un type monomorphe et délègue ce calcul à une fonction spécialisée quand elle rencontre un type paramétré.

Comme la reconstruction d'un type peut contenir des types indéfinis (`undef`), le résultat de `type_of_val` peut être imprécis par rapport au type attendu d'une valeur. Néanmoins cette reconstruction peut être utilisée pour explorer ou afficher des valeurs grâce à cette reconstruction partielle. Pour les types récursifs il faut garantir que cette exploration ne boucle pas. Si on déclare le type `z` et la valeur `x` de la manière suivante, on obtient une valeur circulaire.

type récursif

```
type 'a z = K of 'a z;;
let rec x = K x;;
```

Pour éviter d'explorer indéfiniment l'algorithme proposé doit connaître les cellules explorées soit en marquant leur pointeur, soit en construisant une liste d'association (ou table de hachage).

Cette fonction de reconstruction de types n'explore pas les valeurs fonctionnelles et ne détecte pas les variables de type faibles. Mais il est déjà possible d'afficher l'environnement d'une fermeture. En résumé, on garde la trace du passé, on explore le présent mais on ne peut pas inférer le futur. C'est-à-dire, on ne peut pas calculer le type précis de valeurs fonctionnelles.

Valeurs fonctionnelles

Pour reconstruire un type fonctionnel, il devient nécessaire d'ajouter une information sur le typage statique pour toutes les λ-abstractions d'un programme. On tente alors d'utiliser conjointement l'information issue du typage avec les informations de type extraites de l'environnement pour déterminer le type des valeurs fonctionnelles rencontrées durant l'exécution.

Soit le programme suivant :

un exemple fonctionnel

```
let proj = function x -> function y -> x;;
let g  = proj 3;;
```

Deux types statiques sont associés aux deux abstractions λx et λy. Le premier est $\alpha \rightarrow \beta \rightarrow \alpha$ et le deuxième $\beta \rightarrow t_x$; où t_x est le type de x dans l'environnement de typage. Cela autorise de reconstruire le type d'une application partielle comme `g` seulement à partir d'informations de type de valeurs existantes sans avoir besoin de propager dynamiquement l'information de type quand une fonction est appliquée. Dans notre exemple, on obtient $\beta \rightarrow int$. La valeur de t_x est ici facile à déterminer à partir de la valeur de `x` de l'environnement de la fermeture `g`. En règle générale le calcul est plus complexe pour la détermination du type d'une fermeture à partir de son environnement.

Dans ce cas aussi le type reconstruit peut être moins précis que celui attendu. Cela est d'autant plus vrai en présence de variables de type faibles. En Objective Caml, l'inférence de types ne généralise pas les variables de type d'expressions *expansives*, incluant l'application. Dans l'exemple suivant, il y a une différence entre les types de `r` et de `rt` :

variables de type faibles

```
# let make_p u = u,u;;
make_p : 'a -> 'a * 'a
# let id x = x;;
id : 'a -> 'a
# let r = make_p id;;
r : ('_a -> '_a) * ('_a -> '_a)
# let rt = id,id;;
rt : ('a -> 'a) * ('b -> 'b)
```

Mais ces deux valeurs correspondent à des allocations équivalentes dans le tas. Aussi notre reconstruction de type trouvera les mêmes types pour les deux. On perd une partie de l'histoire du calcul.

Limitations

Dans cette proposition on ne conserve l'information de type de la phase d'inférence seulement pour les abstractions (et donc pour les fermetures). Pour cela on peut reconstruire un type contenant plus d'inconnues de type qu'attendues. Dans le premier cas on construit des inconnues de type là où l'on espérait un type monomorphe. Dans le deuxième cas on construit un type contenant plus d'inconnues de par la perte de partage des variables de type (perte liée aux types extraits de l'environnement). Aussi l'utilisation du type reconstruit et de la valeur associée doit être limitée pour éviter de l'employer à d'autres fins que celle initialement prévue au typage du programme. C'est une restriction pour les outils de mise au point (quand on veut appliquer une fermeture interactivement à un argument construit à la main) et pour la persistance des valeurs. Dans ces deux cas il est nécessaire de limiter leur usage à des types reconstruit monomorphes.

4.4 Conclusion : environnement portable

De nombreux programmeurs gardent un souvenir ému de l'environnement de développement de Turbo-Pascal. Sa prise en main était rapide. L'interaction entre l'éditeur structuré et la compilation ou la trace d'exécution étaient facilement compréhensibles pour le débutant. Certes il ne fonctionnait que pour une plate-forme de développement et paraîtrait probablement archaïque au programmeur d'aujourd'hui. Néanmoins son adéquation entre les besoins du programmeur et la technologie de l'époque a garanti son

succès.

On demande plus de choses aujourd'hui à un environnement de développement tout en désirant une prise en main rapide. Pour le langage CAML il me parait nécessaire d'effectuer des liens entre un éditeur structuré et d'une part le compilateur pour le typage et d'autre part avec un outil de mise au point à l'exécution.

L'information sur les types doit être à tout moment accessible y compris pour les déclarations locales. La nouvelle distribution d'Objective Caml permet d'afficher l'arbre de syntaxe typé d'un programme (option `-dtypes` du compilateur). Celle-ci peut alors être utilisée dans `ocaide` pour une navigation typée dans l'éditeur.

La reconstruction dynamique de types, même partielle, est une brique essentielle à l'exploration des valeurs durant l'exécution d'un programme. Mais elle nécessite des modifications profondes du compilateur pour la représentation des données et le GC. La nouvelle plate-forme d'exécution .NET a besoin d'une transmission de l'information de types calculée à la compilation pour améliorer le code produit typé. Une voie intéressante est d'affiner cette information pour la rendre utilisable par les outils de mise au point de cette plate-forme.

Par ailleurs l'expérience `ocaide` a montré que l'on pouvait construire et utiliser un *IDE* portable. Si les portages de la plate-forme .NET tiennent leurs promesses, alors il est possible d'intégrer dans un *IDE* à la `ocaide` une version du compilateur O'CAMiL produisant un code qui inclut des informations de types pour les outils de mise au point.

Chapitre 5

Enseignement

La difficulté d'apprentissage des langages fonctionnels typés statiquement est un sujet récurrent sur les listes de diffusion. La dernière vague sur la *caml-list* date du printemps 2003 et a produit en quelques jours une centaine de transactions. De ces nombreux messages deux citations opposées reflètent la teneur des discussions. La première est extraite de «Patterns of Software»[1] de Richard Gabriel[66] :

> « *The second mandatory feature is that the language cannot require mathematical sophistication from its users. Programmers are not mathematicians, no matter how much we wish and wish for it. And I don't mean sophisticated mathematicians, but just people who can think precisely and rigorously in the way that mathematicians can.*
>
> *For example, to most language theorists the purpose of types is to enable the compiler to reason about the correctness of a program in at least skeletal terms. Such reasoning can produce programs with no run-time type errors. Strict type systems along with other exemplary behavior, moreover, enable certain proofs about programs.*
>
> *Well, the average programmer might be grateful somewhere in his heart for the lack of run-time type errors, but all he or she really cares about is that the compiler can produce good code, and types help compiler writers write compilers that do that.*
>
> *Furthermore, types that try to do more than that are a hindrance to understanding. For example, abstract data types allow one to write programs in which the interface is completely separate from the implementation. Perhaps some large programming teams care a lot about that, but a lot of programming goes on in ones and twos, maybe threes. Types of this variety, with complicated syntax and baffling semantics, is a hindrance to programming for many of these folks. Sure, someday this will be important as the size of programming teams increases and as the level of mathematics education increases (no laughing, please), but today it is not.*
>
> *Inheritance and polymorphism are examples of programming language concepts requiring mathematical sophistication to understand.*»

Et la deuxième est due à Robin Milner préfaçant «The Little MLer»[58] :

> « *This is a book about writing programs, and understanding them as you write them. Most large computer programs are never completely understood ; if they were, they wouldn't go wrong so often and we would be able to describe what they do in a scientific way. A good language should help to improve this state of affairs.*
>
> *There are many ways of trying to understand programs. People often rely too much on one way, which is called "debugging" and consists of running a partly-understood program to see if it does what you expected. Another way, which ML advocates, is to install some means of understanding in the very programs themselves.* »

[1] pages 128-129 du chapitre «The End of History and the Last Programming Language».

Comme indiqué dans l'introduction les attentes vis-à-vis d'un langage proviennent de la volonté d'abstraire qui de notre point de vue aide à la compréhension des programmes. Néanmoins l'argumentaire de Gabriel n'est pas à sous-estimer dans la mesure où l'enseignement de la programmation incluant les langages fonctionnels (typés statiquement) veut s'adresser au plus grand nombre. L'approche de Milner est bien adaptée à la formation du spécialiste de la discipline. La difficulté provient alors de l'adaptation aux cursus académiques et à la formation permanente de cette approche réfléchie de la programmation. En étudiant l'excellent rapport[148] de l'ACM[2] sur les cursus en «*Computer Science*», on peut noter l'importance de la programmation sous ses différents aspects et les multiples implantations possibles de ceux-ci. Qu'en sera-t-il de la réforme imminente dite «LMD»[3] des cursus universitaires ?

Notre propos dans ce chapitre est de tenter de répondre à la critique courante de l'abord difficile des langages fonctionnels statiquement typés. La réponse que nous développons repose principalement sur l'adaptation de la présentation des concepts selon le public tant universitaire que professionnel.

Pour cela nous présentons plusieurs implantations de l'enseignement de ML selon les niveaux y compris comme premier langage de programmation pour les débutants. Pour accentuer le côté généraliste du langage nous montrons son utilisation comme langage support à un nombre important d'autres cours d'informatique. Nous explorons ensuite son intérêt pour d'autres disciplines. Notre expérience de formations courtes du langage Objective Caml à des publics d'ingénieurs permet d'exprimer leurs demandes. Enfin nous nous intéressons aux outils d'aide à la formation. L'informatique et en particulier la programmation se prête bien aux TICE[4]. Les expérimentations récentes effectuées à l'université Pierre et Marie Curie (Paris 6) permettent d'appréhender la manière dont les étudiants abordent ces nouveaux outils comme par exemple les examens sur machine avec corrections complètement automatisées.

5.1 Enseigner ML

Enseigner un langage n'est pas une fin en soi. L'enseignant cherche à faire passer des concepts plus ou moins abstraits pour le domaine de son cours comme par exemple l'écriture d'algorithmes, la construction d'applications ou les techniques de compilation des langages. On essaie, en déclinant plusieurs variations de cours existants, de déterminer la manière dont ML peut prendre sa place. Un certain nombre d'arguments utilisés ici peuvent être repris pour les langages fonctionnels typés dynamiquement comme Scheme ou pour les langages fonctionnels purs à la Haskell. On présente tout d'abord l'approche classique où à partir de la compréhension d'un modèle de calcul (le λ-calcul) on passe par ajouts successifs à un langage de programmation complet. La deuxième approche est orientée «types et structures de données». Là aussi on construit à partir de structures simples les structures de données classiques dans un cadre typé statiquement. La troisième approche concerne l'organisation et la structuration d'applications. Elle compare les modèles objets et modules paramétrés. La quatrième approche concerne les débutants et s'interroge sur l'apprentissage de ML comme premier langage.

5.1.1 Du λ-calcul aux langages fonctionnels

Le λ-calcul donne une définition formelle de ce qu'est un calcul. Il permet de raisonner sur les calculs, sur les stratégies d'évaluation des calculs et sur la terminaison d'un calcul. Ce modèle de calcul est sous-jacent pour les langages fonctionnels.

La présentation du modèle, précédant celle d'un langage, est une approche classique qui est souvent proposée en maîtrise ou en DEA en particulier pour l'introduction des types et propriétés sur les types. On simplifie ainsi les langages pour se concentrer sur leurs modèles.

[2] Association for Computing Machinery (`www.acm.org`).
[3] Licence-Master-Doctorat.
[4] Technologies de l'Information et de la Communication pour l'Enseignement.

C'est aussi de notre point de vue la possibilité de bien appréhender les caractéristiques des langages fonctionnels que sont : les stratégies d'évaluation (immédiate ou retardée) et les valeurs fonctionnelles. L'écriture de structures de contrôle comme la récursion s'exprime aussi comme un simple combinateur (de point fixe) selon la stratégie employée. C'est aussi l'occasion de coder différentes structures de données (booléens, entiers, couples) par des λ-termes correspondant à des valeurs fonctionnelles. On apprécie ainsi qu'un programme soit une donnée et que les données puissent être codées par des programmes.

L'introduction d'un système de types, comme les types simples, apporte d'une part une information utile au calcul et d'autre part facilite le raisonnement sur la terminaison des programmes. On introduit aussi la nature du polymorphisme à la ML dans ce cadre simplifié.

Le passage du λ-calcul à un «vrai» langage s'effectue alors par l'adjonction de types et structures de données de base et de «formes spéciales» (structures de contrôle ne suivant pas la stratégie établie). Cela est d'autant plus nécessaire pour les langages à évaluation immédiate comme Scheme ou ML. On comprend plus facilement ainsi pourquoi la récursion doit être considérée comme une forme spéciale et pourquoi elle fait perdre la propriété de terminaison des programmes typés.

Ce type de présentation peut être prodigué à des étudiants suivant une voie professionnalisante comme les DESS. L'introduction formelle aux systèmes de types peut ensuite être utilisée pour comparer les systèmes de types de langages plus industriels comme Java. Cette voie est suivie depuis plusieurs années au DESS GLA[5] dans le cours «Langages Fonctionnels et Typage»[6].

5.1.2 Types et structures de données

Le cours «types et structures de données» nous semble un des cours incontournables pour la formation d'informaticiens. Il est bien sûr orienté algorithmique et programmation. Il permet de construire à partir des éléments de base d'un langage les principales structures de données que les étudiants étudieront par la suite dans des cours plus spécialisés. On peut citer d'une part les structures linéaires (listes, tableaux, tables de hachage, ...) et d'autre part les structures arborescentes (arbres binaires, n-aires) et les structures cycliques (automates, graphes).

L'usage d'un langage statiquement typé aide à la manipulation des opérateurs ou fonctions sur ces structures. Ainsi la déclaration d'une nouvelle structure entraîne la définition d'un type, des fonctions de création et d'accès puis des opérateurs sur ces valeurs de ce type comme dans [113].

Toutes les structures de données classiques ne font pas toujours bon ménage avec le style purement fonctionnel. Pour cela un langage fonctionnel et impératif est nécessaire. Les structures comme les tableaux ou les structures cycliques s'expriment de prime abord plus facilement en impératif. Un deuxième intérêt est de pouvoir coder de deux façons très différentes une même structure de données.

Cela amène naturellement aux types abstraits de données où l'on masque l'implantation d'une structure de données. Le type est connu mais pas sa définition. Cela force l'étudiant à prendre du recul en exprimant ses besoins (éléments et opérateurs) avant de passer à leur réalisation. Cette nécessaire spécification passe par l'écriture de l'interface d'un module simple Objective Caml. On peut affiner ce masquage avec les déclarations *private*[7] introduites dans la dernière distribution d'Objective Caml.

Ce cours utilisait à la décade précédente le langage Pascal comme langage algorithmique. ML peut alors être considéré comme un super-Pascal de part le polymorphisme paramétrique qui permet de construire des itérateurs fonctionnels plus généraux.

Le nouveau cours[8] de deuxième année de l'actuel DEUG MIAS de Paris 6 est en une illustration. On trouve aussi en classes préparatoires une telle présentation.

[5] Génie des Logiciels Applicatifs.
[6] `www.pps.jussieu.fr/~emmanuel/Public/enseignement/gla_2003_index.html`
[7] *private types* : types de données concrets avec constructeurs et labels privés.
[8] `www.infop6.jussieu.fr/deug/2003/mias/mias-24-b/public`

5.1.3 Modules et Objets

Une autre approche s'intéresse aux niveaux de structuration d'applications dans le cadre du typage statique. Pour cela Objective Caml est un bon candidat. Il permet de bien distinguer la nature de la structuration en classes de celle utilisant les modules paramétrés (par d'autres modules). Cette dernière approche s'inspire des types abstraits de données et pointe les difficultés liées au partage de types. Cette présentation indique les deux voies d'extensibilité : extensions des données et extension des traitements, et souligne leur dualité. Il s'illustre par des exemple de structurations mixtes (objets et modules paramétrés) comme présenté dans [7] et dans [127] dont une version en ligne est accessible [9]. Une version plus spécialisée[10] est aussi en ligne.

Ce cours s'adresse à des étudiants confirmés du niveau maîtrise ou troisième cycle. Il peut faire partie d'un cours sur la programmation objet pour la comparaison des modèles objets dans un but de biodiversité ou dans un cours sur les traits avancés de structuration, s'intégrant alors dans une optique «génie logiciel».

Cette présentation a été donnée pendant cinq ans dans le cours de maîtrise POD[11] et se décline en partie dans le cours actuel MLO[12].

5.1.4 ML comme premier langage

L'influence de l'apprentissage du premier langage a toujours été grande. Elle façonne la vision de l'étudiant sur l'écriture de programmes et son rapport avec la machine.

L'intérêt du modèle fonctionnel comme premier langage est de faciliter la compréhension de plusieurs notions importantes comme la notion de calcul, la composition de fonctions, la récursivité, la création de valeurs et les structures de données dynamiques. Le modèle impératif précise l'utilisation mémoire (partage ou copie) ainsi que le contrôle de l'exécution (séquence, itération). L'utilisation d'un langage possédant ces deux modèles permet de passer d'un haut niveau d'abstraction (fonctionnel) au contrôle plus précis de l'exécution et de l'utilisation mémoire (impératif). Un tel langage permet alors de comparer les deux modèles en particulier entre copie et partage, et entre récursion et itération. Le cadre du typage statique offre un garde fou pour les expérimentations du débutant. Dans ce cadre ML est un bon candidat.

éloignement de la machine Une des critiques principales de l'utilisation de ML comme premier langage provient de son éloignement de la machine.

Une première critique provient de la manipulation mémoire qui est de plus haut niveau qu'en C ou Pascal. Seules les fonctions d'allocation sont visibles (un *cons* (::) sera explicite), les fonctions de récupération mémoire n'existent pas car celle-ci est automatique (GC). Cette critique a été balayée par le succès du langage Java qui a promu la récupération automatique de mémoire comme gage de sûreté d'exécution.

Une deuxième critique porte sur l'abstraction du contrôle de l'exécution : . la notion de fonctions en tant que valeurs manipulables permet de paramétrer les programmes par d'autres programmes. Cette généralisation rend le code écrit plus abstrait. C'est de notre point de vue un grand avantage. Le code sera plus concis et plus lisible. La composition de fonctions est une notion que les étudiants comprennent bien et vite.

De plus la possibilité d'utiliser le style fonctionnel ou le style impératif dans un même langage permet d'adapter l'écriture d'algorithmes dans le style adéquat.

langage algorithmique ML peut être vu comme un langage algorithmique. Il se prête facilement à différents styles de programmation : fonctionnelle, impérative et mixte. Dans [152] et [132] qui sont respectivement deux ouvrages sur l'algorithmie et l'algorithmie graphique, ML est utilisé dans un style impératif principalement comme un super-Pascal. Bien que pour un programmeur fonctionnel cela soit un peu dommage, il est en fait assez important de recouvrir au moins tout Pascal pour un langage algorithmique se voulant général. Dans [49] et [116] l'approche fonctionnelle de la programmation est décrite. Enfin dans [157] les deux styles

[9] `pauillac.inria.fr/~remy/cours/appsem`

[10] `cristal.inria.fr/~remy/poly/mot/`

[11] Programmation Objet et Distribution (`www.pps.jussieu.fr/~emmanuel/Public/enseignement/pod_2001_index.html`).

[12] Modèles et Langages à Objets (`www.infop6.jussieu.fr/maitrise/2003/algoprog/mlo/public/`).

sont présentés par une suite d'exercices conséquents. Il existe d'autre part des ouvrages d'algorithmique utilisant ML comme langage d'implantation comme [126].

expérimentations De nombreuses expérimentations d'enseignement de ML ont été menées dans les premières années universitaires, au CNAM[13] et dans les classes préparatoires. Il n'existe pas de carte nationale des différents langages enseignés dans le labyrinthe des formations supérieures. À l'occasion du *workshop* FLIC[14] Guy Cousineau[47] a effectué un recensement partiel[15] de l'enseignement des langages fonctionnels comme premier langage. Le CNAM a expérimenté l'apprentissage de ML et Ada dans son cycle A[78]. Bien que national, l'implantation régionale de cet enseignement peut revêtir des formes diverses en laissant un choix important aux équipes pédagogiques locales. Les classes préparatoires laissent aussi le choix entre différents langages algorithmiques comme Pascal ou ML, mais dans la très grande majorité des cas c'est Caml-Light qui est utilisé. Globalement ces expériences ont été et sont réussies tant que leurs initiateurs restent en place.

et Scheme La plupart des arguments utilisés pour valoriser le modèle fonctionnel comme premier langage sont valables pour le langage Scheme tant du point de vue abstraction fonctionnelle que comparaison des styles fonctionnel et impératif. L'exemple du DEUG MIAS de Paris 6 est significatif. Le premier cours d'informatique, nommé «Programmation récursive»[16], est basé sur Scheme et l'environnement DrScheme[17][59]. À la sortie de ce cours, les étudiants ont compris la récursion et la manipulation de structures linéaires et arborescentes dynamiques. On peut observer que pour spécifier un problème ou des exercices, un pseudo-langage de types est utilisé. Les étudiants prennent ainsi l'habitude de commenter leur production par une information de types.

une vraie difficulté : l'inférence de types L'intérêt de CAML peut être de nouveau souligné grâce à son typage statique qui allié à l'inférence de type autorise une écriture de programmes aussi concise qu'en Scheme. L'inférence de types est un atout mais aussi une faiblesse dans la mesure où un échec de l'inférence peut produire un message d'erreur difficile à interpréter par le débutant. Il est possible d'expliquer le mécanisme d'inférence mais le détailler n'est pas du niveau de l'initiation d'un langage. La communauté Haskell a réalisé l'environnement Helium[83][18], pour une prise en main facile du langage en explicitant les erreurs dues au typage. Pour cela le typeur de Helium annote l'inférence de type[82] pour produire un message plus précis. Pour ce faire les auteurs ont étudié un large ensemble d'erreurs en journalisant les sessions des étudiants. Ils ont par la suite implanté des heuristiques pour la compréhension de ces erreurs et la production d'une explication plus appropriée. Bien que l'effort soit intéressant et louable il ne nous semble pas forcément judicieux d'essayer de comprendre la volonté du programmeur pour tenter de lui indiquer ses erreurs. L'entremêlement d'erreurs peut être difficile à détecter. Les messages alors obtenus ne sont pas forcément beaucoup plus clairs pour un débutant.

Objective Caml a maintenant un système de types complexe de par l'introduction des objets. et des *variants* polymorphes. On peut le limiter à son noyau fonctionnel et impératif, mais l'on rencontre déjà des difficultés liées au typage des traits impératifs [159]. La non généralisation de l'application fait échouer l'inférence de types de programmes fonctionnels tout à faits corrects sans l'introduction des variables de type faibles. D'autres systèmes permettent de traiter les traits impératifs sans limiter la partie fonctionnelle, comme décrits dans [57], mais leur implantation est fort complexe et pourrait nuire à l'efficacité du typeur.

Peut-être faut-il alors reconsidérer l'inférence de types pour les débutants et ne conserver que le typage statique. Au programmeur de nommer les types de chaque déclaration comme les fonctions et ses paramètres,

[13] Conservatoire National des Arts et Métiers.
[14] *Functional Languages in the Introductory Computing Curriculum.*
[15] `www.pps.jussieu.fr/~cousinea/LF_ENS/LF_ENS.html`
[16] `www.infop6.jussieu.fr/deug/2003/mias/mias-a/public/index.php`
[17] `www.drscheme.org/`
[18] `www.cs.uu.nl/helium`

ou les variables locales. On perd une partie de l'élégance du langage mais le fait d'écrire soi-même les types force le programmeur à une certaine attention et simplifie la vérification du programme. On peut alors plus facilement relier les messages d'erreur aux types définis par le programmeur. Nous y reviendrons dans la conclusion de ce mémoire.

de l'importance de l'environnement Il manque d'autre part un environnement convivial pour la prise en main du débutant à la manière de DrScheme ou d'Helium. En fait sur les plates-formes Unix (Linux, MacOSX) l'utilisation d'Emacs[19] lançant le *toplevel* en interne est presque confortable. Il manque une exécution pas à pas permettant de suivre l'exécution du programme. Par contre sous Windows l'absence d'un environnement léger acceptant le copier/coller se fait cruellement sentir. Comme ce système est bien souvent utilisé pour les cours débutants ou les tutoriaux, un effort est à effectuer de la part de la communauté Objective Caml.

5.2 ML comme langage support

Les cursus universitaires en informatique contiennent un bon fond classique. On retrouve un certain nombre de modules équivalents dans à peu près tous les établissements. Par exemple les cours d'algorithmique, de compilation, de sémantique et preuves, de génie logiciel, d'architecture matérielle, de systèmes d'exploitation, de parallélisme, d'infographie, de bases de données, d'IA, ...peuvent être considérés comme vus dans une licence ou un master d'informatique. À cela se rajoute des cours plus particuliers, comme le temps réel, la programmation par contrainte, etc.

On découpe ces différents cours en trois parties. La première correspond aux cours où ML s'intègre facilement. La deuxième contient les cours où ML se doit d'intégrer de nouvelles fonctionnalités pour pouvoir y répondre. La troisième décrit les cours où ML n'a que peu de points communs et où l'intérêt de l'utiliser semble faible.

5.2.1 Domaines où ML s'intègre

Les quatre domaines suivants : algorithmique, compilation, concurrence et génie logiciel sont propices à l'utilisation de ML.

Algorithmique

Si l'on se réfère à des ouvrages classiques d'algorithmique comme [5] ou [143] il est relativement facile de présenter les différents modèles de données décrits dans cet ouvrage en ML, soit de manière impérative comme dans leur présentation originale soit de manière fonctionnelle. Par exemple une structure de graphes peut être réalisée de manière impérative soit par un tableau d'adjacence, soit par un type récursif mutable :

graphes - version 1

```
# type 'a graph =
    Nil |
    Node of 'a * 'a graph list ref;;
Type graph defined.
# Node (3.14, ref []);;
- : float graph = Node (3, ref [])
```

graphes - version 2

```
# type  cout  = Nan | Cout of float;;
# type  mat_adj =  cout array array;;
# type 'a graphe = { mutable ind : int;
                     taille : int;
                     sommets : 'a array;
                     m :  mat_adj };;
```

ou de manière fonctionnelle en utilisant une liste d'association qui indique les successeurs d'un nœud :

graphe - version 3

```
# type 'a g == (int * 'a  * 'a g list);;
Type g defined.
# ((1,3.14,[]) : float g);;
- : int * float * float g list = 1, 3.14, []
```

La simple copie de graphe diffère de manière importante selon la représentation choisie.

[19] www.gnu.org/software/emacs/emacs.html

Compilation

C'est probablement le cours où l'utilisation d'Objective Caml comme langage d'implantation est la plus intéressante. ML est particulièrement agréable pour la manipulation d'arbres. La construction d'un arbre de syntaxe abstraite, son évaluation ou ses transformations sont alors concises. Pas besoin de modèle objet «Visiteur» [67] pour de tels calculs. Objective Caml propose dans sa distribution des outils pour la construction de grammaires. Le couple `camllex` et `camlyacc` permette de définir des analyseurs ascendants (LALR(1)[20]) à la manière de `lex` et `yacc`. L'intérêt est alors d'avoir un analyseur typé statiquement. Mais Objective Caml propose aussi la construction d'analyseurs descendants en utilisant l'outil `camlp4` pour des classes d'analyseurs HOLL[21]. De plus le modèle fonctionnel permet de construire durant l'analyse syntaxique d'un texte des fonctions d'analyse de la suite de ce texte. On peut alors définir des grammaires contextuelles. Ces analyseurs sont aussi utilisés pour les afficheurs[105]. Il est intéressant de pouvoir comparer ces différents types d'analyseurs.

Plusieurs cours de l'université Paris 6 utilisent d'ailleurs Objective Caml pour cela. Le cours optionnel «Projet» du DEUG MIAS propose pour deux groupes des projets nécessitant une analyse syntaxique : le projet «interprète BASIC» nécessite de construire un analyseur syntaxique pour un noyau de Basic ; le projet «Système de Gestion de Bases de Données Réparties» utilise un analyseur descendant pour un mini-SQL. Le cours de Compilation de Licence a pris CAML pendant plusieurs années comme langage support ; Objective Caml était enseigné auparavant dans le cours de «Programmation» de licence. La modification de ce dernier a entraîné l'abandon d'Objective Caml au profit d'Ada et de C.

On peut comparer l'intérêt de ML et de C comme langage support en lisant les livres d'Andrew Appel sur les compilateurs modernes [12, 11], qui montrent une implantation en ML et une autre en C (une troisième version utilise Java). Le cours «Compilation Avancée» de maîtrise utilise Objective Caml comme langage support.

On trouve de nombreux supports de cours utilisant Objective Caml ou Standard ML dans des cours introductifs et avancés. En voici quelques exemples : les cours de compilation de l'école polytechnique «Langages et Compilation» de Didier Rémy[22] et «Compilation» de Luc Maranget[23], le cours de compilation de maîtrise de Paris 7 de Roberto DiCosmo[24], le cours d'Olivier Danvy[25] à l'université d'Aarhus. Le nombre et la qualité des ressources facilement accessibles montre bien un intérêt toujours fort des enseignants pour l'utilisation de ML dans leur cours de compilation.

Concurrence

Dans le cours PC2R[26], toujours en maîtrise de Paris 6, nous comparions les mécanismes de concurrence dans les langages Objective Caml et Java. L'intérêt d'utiliser Objective Caml est de montrer en action deux modèles de concurrence : le modèle de mémoire partagée réalisé à base de sémaphores pour des processus légers et un modèle de mémoire distincte avec synchronisation directement inspiré de CML[129]. Ce dernier modèle permet d'effectuer le passage au modèle à mémoire répartie illustrée par les prises de communication (*sockets*) des protocoles UDP/IP et TCP/IP du réseau Internet. De même les nombreuses extensions parallèles de ML peuvent être comparées (voir page 106).

Génie logiciel

Bien que les rapports entre génie logiciel (*Software Engineering*) et science informatique (*Computer Science*) sont toujours d'actualité [115], on rencontre le plus souvent un tel cours dans un cursus univer-

[20] *Lookahead Left to Right Parsing.*
[21] *High Order LL (Left-to-right parse, Leftmost derivation).*
[22] `cristal.inria.fr/~remy/poly/compil/`
[23] `pauillac.inria.fr/~maranget/X/compil/`
[24] `www.pps.jussieu.fr/~dicosmo/CourseNotes/Compilation`
[25] `www.daimi.au.dk/~eernst/icvm03/`
[26] Programmation Concurrente Réactive et Répartie (`www.infop6.jussieu.fr/maitrise/2003/algoprog/pc2r/public/`).

sitaire d'informatique. Une part importante du génie logiciel se base sur les types abstraits de données et sur les systèmes de modules. Dans [80] ces thèmes sont largement décrits. Il semble clair que le système puissant de modules de SML/NJ ou d'Objective Caml est propice à la construction modulaire générique d'applications. Ce qui peut être résumé par le slogan «*Higher-order + Polymorphic = Reusable*» de [149].

Peu de méthodologies de développement ont été proposées pour les langages fonctionnels, on peut citer [131] qui définit une *Functional Analysis and Design Methodology* (*FAD*) introduisant une notation graphique pour la composition et pour la modélisation. *FAD* est orienté langage fonctionnel pur statiquement typé à la Haskell.

5.2.2 Quand ML a besoin d'intégrer d'autres traits

Pour une autre classe de cours, plus dépendant de l'interopérabilité du langage, comme les cours de systèmes d'exploitation, de parallélisme, d'infographie, de programmation avec contraintes, ML n'est pas facilement utilisable sans de nouvelles bibliothèques ou de nouvelles extensions du langages, qui doivent garder les propriétés de typage de ML. Plusieurs des extensions présentées ne s'intègrent pas complètement dans un système de types existant.

Programmation parallèle

ML n'a pas été conçu initialement pour le parallélisme ou la répartition, néanmoins une trouve une grande richesse d'extensions. La principale difficulté provient des contraintes de typage de la communication. On trouve différentes extensions de ML qui intègrent un de ces paradigmes comme Facile [70], Conccurent ML [129], Distributed ML [93], paraML [17], LCS [19], Caml-FLight [62], BSMLIB[77], OCamlP3L[51], JoCaml[45].

On peut décrire rapidement les différences de ces systèmes :

Facile, CML, LCS : parallélisme de contrôle à grain fin, mémoire partagée, création de processus, événements de synchronisation (description comportementale inspirée de CCS pour LCS et Facile) , permet l'écriture de protocole,

paraML : inclut la création de processus pour supporter un parallélisme à grain moyen,

Caml-FLight : parallélisme de données à gros grain, mémoire distribuée, pas de création de processus, mécanisme d'évaluation à distance, basé sur un protocole.

BSMLIB : [77] : pour un calcul du coût[27] ;

OCamlP3L : : pour la composition de squelettes parallèles[28] ;

JoCaml : pour la mobilité[29].

Toutes ces approches ont peu en commun. Elles permettent de dresser un panorama assez important des extensions parallèles de ML et de les comparer.

Graphisme et interfaces

Ce cours se découpe en général en deux parties : la première concerne les modèles graphiques et la deuxième les interfaces graphiques. Dans le premier cas l'interaction n'est pas obligatoire. On trouvera dans [35] une description d'une bibliothèque graphique pour la construction d'illustrations implantant dans un style fonctionnel le modèle graphique de PostScript. L'intérêt ici n'est pas juste de décrire des algorithmes graphiques en ML impératif comme dans [132] mais de montrer l'apport de la programmation fonctionnelle dans la programmation graphique.

[27] `f.loulergue.free.fr/`

[28] `www.pps.jussieu.fr/~dicosmo/ResearchThemes/PARA/OCAMLP3L/qui.di.unipi.it/index.html`

[29] `pauillac.inria.fr/jocaml/`

Objective Caml est distribué avec deux bibliothèques graphiques `graphics` et `camltk` fonctionnant sous différents systèmes de fenêtres (Windows, MacOSX, et X-window). `graphics` est assez minimaliste, nous y reviendrons au chapitre 6. La boîte à outils (libRT) est un outil de plus haut niveau basé sur la Xlib qui peut être utilisée pour présenter le système X-window. De même la bibliothèque eXene distribuée avec Concurrent ML est elle aussi un candidat à la construction d'interface au dessus de X. Son intérêt provient de la création de processus légers pour les fenêtres qui donne les bons outils pour construire une interface concurrente.

Système

Le cours sur les systèmes d'exploitation décrit de manière précise le noyau d'un système, en général le système Unix. Il est clair que ML n'a pas été conçu pour cela, mais rien ne l'empêche de l'interfacer avec la bibliothèque système du cours comme par exemple en [95]. Ce polycopié présente la programmation du système Unix à partir de Caml-light. Un cours sur le système d'exploitation Unix a été donné à l'école normale supérieure par Xavier Leroy en Caml-Light et en C à l'ENS. Didier Rémy[30] base son enseignement système à l'école polytechnique sur Objective Caml

Le projet FoxNet[31][79] est une illustration conséquente de programmation système pour l'écriture de construction de systèmes d'exploitation en particulier du protocole TCP/IP et d'applications réseau.

Il est clair néanmoins que ML a quelques lacunes de bas niveau pour une programmation système (comme par exemple les champs de bits). Par contre l'intérêt de ML dans la construction de système vient principalement de son système de types (les extensions sont sûres du point de vue des types) et de sa modularité.

Programmation par contraintes

Ce type de programmation déclarative est issu de la programmation logique en remplaçant l'unification par la résolution de contraintes sur un domaine particulier. Néanmoins on peut s'affranchir du coté programmation logique pour enseigner à part entière la partie résolution de contraintes. Une extension de programmation par contraintes sur les domaines finis a été proposée dans [34]. Cette extension s'intègre sous forme de bibliothèque en ML. Elle permet de combiner dans un seul programme le style fonctionnel et le style de programmation par contraintes. Une bibliothèque plus récente et plus riche, due à Pascal Brisset[32][14], facilite ce style de programmation.

5.2.3 Où ML apporte peu

Il reste néanmoins certains cours, nécessitant une part non négligeable de programmation où ML s'intègre mal. Premièrement quand le thème du cours est un autre paradigme de programmation et dans ce cas autant utiliser le langage approprié. C'est le cas de la programmation logique. Les expériences d'intégration de programmation logique en CAML nécessitent une refonte du langage. Dans ce cas il existe d'autres alternatives comme le langage Mercury[141].

Deuxièmement s'il est nécessaire, comme dans un cours de matériel, de se rapprocher de manière fine des machines en passant par la programmation assembleur. Troisièmement quand il est obligatoire d'avoir une gestion précise du temps comme dans la partie temps réelle du contrôle de processus (au sens industriel) ce qui ne correspond pas à un langage où la récupération de données (GC) est automatique et où son déclenchement est imprévisible. Le fait de majorer par le pire temps du GC est difficilement acceptable dans ce cas là.

5.3 ML pour les autres disciplines

Dans d'autres disciplines l'enseignement de l'informatique se pratique par une initiation d'un langage de programmation et des bases de l'algorithmique. Il est intéressant de montrer les apports de ML dans

[30] `cristal.inria.fr/~remy/poly/system/`

[31] `foxnet.cs.cmu.edu`

[32] `www.recherche.enac.fr/opti/facile`

ces disciplines en particulier par son système de types. On s'intéressera tout particulièrement aux sciences, proches de l'informatique, comme les mathématiques et la physique.

Mathématiques Le langage ML est particulièrement apprécié des mathématiciens pour les raisons suivantes : la notion de fonction est bien connue, la manipulation des fonctions comme valeur est aussi naturelle (la construction d'ensemble de fonctions va de soi en mathématique), la notion de domaine et codomaine dont le type est une approximation est là encore immédiate. En fait l'abstraction de fonctions et leur application sont des concepts déjà utilisés en mathématiques. Le mathématicien spécialisé en logique comprendra aisément le passage du λ-calcul au langage ML.

ML permet la construction d'applications mathématiques, soit en calcul numérique, soit en calcul exact. On trouvera pour SML/NJ une implantation des grands entiers et pour Objective Caml une bibliothèque d'arithmétique entière et réelle en précision arbitraire. Cette dernière peut intéresser l'enseignant de calcul formel qui est selon les cas intégré en mathématiques ou en informatique. Par exemple le système FOC [33] fournit des bibliothèques[24] en Objective Caml pour le calcul formel. Jusqu'à maintenant les langages d'implantation et de programmation de tels systèmes s'effectuait plutôt en Lisp ou C++ comme le système GB[34]

Physique L'enseignement d'un langage de programmtion en Physique concerne bien souvent le langage Fortran (car il existe une base importante de bibliothèques écrites en ce langage) et C. En dehors des qualités déjà décrites du langage ML (sûreté, expressivité,...) le typage de ML étendu aux unités physiques [76] devrait intéresser le physicien que les conversions de types intéressent peu. Il est effectivement bien agréable d'utiliser des fonctions manipulant indifféremment des valeurs d'unités différentes et dont les conversions sont automatiques, comme par exemple calculer sur les degrés Celcius ou Fahrenheit. D'autres intégrations de systèmes d'unités ont été réalisées pour Standard ML [91] ou CAML[35].

Autres disciplines Il est déjà difficile de recenser l'emploi des langages fonctionnels dans les cursus informatiques. Les exemples donnés ci-dessus se cantonnent bien souvent à l'Ile de France. La tâche pour les autres disciplines est encore plus ardue. On peut néanmoins évoquer les expériences effectuées à l'Institut Pasteur pour les biologistes et à l'université Paris 7 pour les linguistes.

Pendant plusieurs années (de 1994 à 1999) le «Service d'Informatique Scientifique» de l'Institut Pasteur a prodigué une formation informatique[36] basée pour la partie programmation sur le langage Scheme pour la résolution de problèmes biologiques. Cette expérience a débouché sur un ouvrage d'initiation[21].

La linguistique a toujours été proche du calcul symbolique, promouvant en cela les langages fonctionnels et objets pour le traitement automatique des langues. L'enseignement du DEA de «Linguistique Théorique, Descriptive et Automatique»[37] le montre. Les projets sont à réaliser en Objective Caml ou Java. Le *toolkit* Zen[38][85] de linguistique computationnelle développé par Gérard Huet est implanté en «Pidgin ML», noyau d'Objective Caml dont la syntaxe est révisée.

5.4 ML en formation permanente

L'enseignement en formation permanente touche un public bien différent des étudiants sur un temps bien plus court. Le public est restreint et motivé par une utilisation pratique du langage dans un futur immédiat. La période d'enseignement dépasse rarement deux semaines d'où une certaine difficulté à l'absorption rapide de nouveaux concepts. Nous avons monté une proposition de stage de Formation Permanente en Objective

[33] `www-spi.lip6.fr/foc`
[34] `fgbrs.lip6.fr/jcf/Software/Gb/index.html`
[35] `www.di.ens.fr/~blanchet/`
[36] `www.pasteur.fr/recherche/unites/sis/Rapport/node5.html`
[37] `talana.linguist.jussieu.fr/enseignement/dea-li.html`
[38] `pauillac.inria.fr/~huet/ZEN/index.html`

Caml à l'université Paris 6[39]. Sur ces dernières années nous avons effectué six stages d'une semaine pour les ingénieurs du CEA et de FT/RD pour des publics d'ingénieurs et de chercheurs de ces compagnies.

5.5 ML et les TICE

Une partie de l'enseignement de la programmation se passe pour l'étudiant face à la machine soit durant les «Travaux sur Machine» soit par les devoirs à rendre. On peut prolonger cette relation quotidienne par des devoirs ou des épreuves en temps limité sur machine. On décrit une expérience de notation automatique de masse utilisée pour la licence d'informatique de Paris 6. La programmation est une des disciplines qui se prête bien à l'auto-formation dans la mesure où un certain nombre d'exercices peuvent être testés immédiatement sur machines. Une section du deug MIAS, appelée SPAD pour le semi-présentiel, a utilisé un cédérom d'auto-formation pour Scheme. Nous le présentons et discutons de la possibilité d'utiliser CAML dans ce cadre.

épreuves sur machines corrigées automatiquement L'intérêt alors est de tester dans des conditions «réelles» comment l'étudiant aborde et implante un problème. Christian Queinnec organise depuis maintenant trois ans de telles épreuves en licence d'informatique. Outre des problèmes de logistique inhérents au grand nombre d'étudiants (environ 400 en licence) ce qui nécessite de trouver suffisament d'ordinateurs disponibles et correctement configurés, la grande difficulté est la notation automatique [124] de ces épreuves. Pour cela un ensemble d'outils, appelé CFSkit[40], est proposé pour tester les épreuves et annoter les corrections. Au concepteur des épreuves de construire les jeux de tests pour l'instauration du barème et les indications sur la copie électronique de l'étudiant. Il est à noter que les étudiants acceptent bien cette notation «automatique» dans la mesure où leur copie est bien annotée, *i.e.* qu'ils comprennent quels tests ont été effectués et comment leurs programmes ont réagi.

Ce contrôle correspond pour chaque semestre à quatre épreuves (un peu moins de cent étudiants par épreuve). Pour s'entraîner les étudiants peuvent rejouer avec les examens passés[41].

Ces épreuves laissent libre le choix du langage d'implantation. Entre les épreuves du premier semestre et celle du second, on s'aperçoit de l'aisance grandissante des étudiants en programmation dans les choix plus variés du ou des langages pour leurs solutions. CAML a été faiblement choisi par les étudiants dans la mesure où ils utilisent fréquemment le ou les derniers langages vus en cours. Néanmoins Objective Caml est un candidat intéressant pour les épreuves à durée limitée par les facilités qu'offre le *toplevel* pour un langage statiquement typé.

auto-formation Le cédérom du cours «Programmation récursive» s'adresse aux étudiants suivant la section de Deug MIAS de Paris 6 semi-présentielle[27]. Ces étudiants ne suivent pas de cours magistral, ils ont par contre des rendez-vous hebdomadaires en demi-groupes avec leurs enseignants. Ce cédérom remplace l'enseignement magistral prodigué habituellement dans des amphithéatres assez chargés (environ deux cents inscrits par section) et en partie les travaux dirigés et pratiques. Suite à la lecture d'un chapitre du cours du cédérom, l'étudiant peut s'exercer et s'auto-évaluer par un ensemble de *quizz* correspondant aux concepts du chapitre. Ces questions de programmation présentées dans un navigateur web nécessitent que l'étudiant programme. Pour cela les quizz sont reliés à l'environnement DrScheme qui évalue à la volée ses réponses. Les solutions sont vérifiées sur des jeux de tests construits par les enseignants. Les erreurs sont reportées visuellement à l'étudiant pouvant alors corriger sa solution erronée.

Le langage de cet enseignement est Scheme mais, si l'on sait communiquer avec le *toplevel* Objective Caml comme présenté au chapitre 3 ce dernier pourrait faire office de langage support pour un tel cours.

[39] `wwwadm.admp6.jussieu.fr/fp/catalogue.htm`
[40] Kit pour le Contrôle Final Semestriel : `www-spi.lip6.fr/~queinnec/Programs/`
[41] Annales CFS : `/www.infop6.jussieu.fr/licence/annales/cfs/`

5.6 Conclusion : et les objets ?

Le colloque scientifique en l'honneur de Jean-François Perrot [20] sur le thème «20 ans après : où en sont les objets» s'est conclu par une table ronde commentant la récente discussion entre Richard Gabriel[64] et Guy Steele[145] sur «les objets ont-ils échoué ?». Christian Queinnec concluait son texte introductif ainsi :

> « *Que les hordes barbares envahissent notre univers de lettrés est donc un progrès puisque plus peuvent programmer et ainsi contribuer à enrichir notre univers même si leur culture n'est pas à la hauteur du raffinement que nous a inculqué Jean-François Perrot. Tout comme il est préférable que tous sachent obtenir une racine carrée (avec une calculette) plutôt que de laisser une seule poignée savoir la calculer (avec un crayon et du papier), il est plus satisfaisant de laisser s'empoussiérer nos disputes et de nous tourner vers les deux problèmes majeurs de l'enseignement suivants :*
> – *Comment enseigner l'objet* ab initio *aux masses ?*
> – *Comment enseigner l'«informatique» aux futurs experts ?*
> *Et ainsi les hordes pourront-elles fonder une nouvelle civilisation.* »

Notre réponse est qu'il n'existe pas (encore) un langage à objets pédagogique, et donc l'enseignement de l'objet aux masses est difficile si l'on cherche à utiliser les langages objets à succès comme Java, C# ou VB.NET. Ces derniers sont souvent considérés comme des langages ATF[42] tant pour la modélisation de système d'informations, que pour la description d'interfaces utilisateurs et même l'écriture d'algorithme. Il nous semble qu'il est plus facile d'aborder l'objet en ayant assimilé ce qu'est un calcul ; l'objet étant alors utilisé pour la modélisation. Si l'on parle de calcul, le modèle fonctionnel est approprié. Si l'on pousse ce raisonnement, l'enseignement de l'objet passe par la programmation fonctionnelle (qu'elle soit typée statiquement ou dynamiquement).

Une fois ces notions de calcul et de composition de calculs bien assimilées il devient possible de comprendre le mécanisme de liaison tardive et l'intérêt de la spécialisation par héritage. Si l'on cherche à sauter cette étape, il est quasiment obligatoire de masquer les spécificités des objets, ce qu'est un point d'entrée d'un programme[43], ce qu'est une instance d'une classe locale pour définir un écouteur sur un composant graphique, etc. On retombe alors dans les «trucs et astuces» de la programmation chers à la presse informatique, ce qui a peu de sens pour un enseignement de base de la programmation.

Pour le spécialiste la réponse est plus facile : il doit connaître les différents paradigmes de programmation séquentielle (impératif, fonctionnel, objets) et leurs systèmes de typage (dynamique ou statique). L'ordre importe peu du moment que ces différents modèles sont vus et comparés.

[42] À Tout Faire.
[43] Le trop célèbre `psvm` pour `public static void main` de Java.

Chapitre 6

Diffusion d'applications

Les langages fonctionnels sont principalement issus du monde académique ainsi que la majeure partie de leur communauté de programmeurs. Ils suivent le modèle «open source» qui favorise la diffusion d'applications par l'échange des sources à recompiler. L'intérêt est de pouvoir connaître ce qui va effectivement être exécuté sur sa machine. En cela c'est un gage de sûreté. Néanmoins les différentes dépendances d'une application avec certaines bibliothèques nécessitent bien souvent de devoir aussi les (re)compiler. Une installation devient rapidement un véritable jeu de piste. Ce modèle s'adresse au programmeur ou du moins à un utilisateur maniant avec dextérité les `makefile` et autres `configure`. Il ne peut pas être étendu à l'utilisateur néophyte.

Pour cela, y compris pour les systèmes «open source» comme Linux ou BSD, il existe des systèmes de paquetages permettant de distribuer des exécutables et d'indiquer les dépendances pour les bibliothèques. Les principaux sont `rpm` pour RedHat/Mandrake (Linux), `package` pour Debian (Linux) et `port` pour Free-BSD (BSD). Comme ces systèmes sont portés sur plusieurs architectures machine, la diffusion d'une application nécessite autant de paquetages que de couples (système,processeur). Le système NextStep qui tournait sur différents processeurs (Motorola 68040, Intel, HP-PA et Sparc) avait à son époque introduit la notion de «gros» binaires (*fat binaries*) que l'on pouvait produire à partir d'une seule plate-forme. Associés à un système de paquetages ils pouvaient alors être distribués pour les différents processeurs en autorisant l'installateur à ne conserver que le binaire utile[1] pour son ordinateur.

Comme la plupart des applications utilise une interface graphique pour interagir avec l'utilisateur (*Graphic User Interface* ou *GUI*) le système de gestion de fenêtre a lui aussi son importance. On trouve principalement les trois systèmes suivants : X-window associé à Unix, Windows et ses dérivés et Carbon-Cocoa pour MacOSX. Il est à noter que la partie interaction d'une application avec l'utilisateur prend une part de plus en plus importante dans la réalisation d'un programme. Le concepteur d'applications a alors le choix entre utiliser une bibliothèque graphique portée sur ces trois systèmes ou bien d'en choisir une en profitant alors de ses spécifiés. On peut citer l'exemple du langage Java qui avec ses bibliothèques AWT[161] et Swing[103] arrive à s'exécuter sur ces trois systèmes de gestion de fenêtres au prix quelquefois d'une perte d'efficacité. Eclipse a développé sa propre bibliothèque graphique Java, appelée SWT[2], pour améliorer l'efficacité quitte à perdre en sûreté d'exécution.

D'autres points sont utiles pour une meilleure diffusion pour le grand public comme la facilité d'installation et de mise à jour, et la régionalisation des applications.

Nous présentons alors nos travaux qui découlent des chapitres précédents. On s'intéresse tout d'abord aux interfaces graphiques portables pour Objective Caml, tout particulièrement la bibliothèque `graphics`. On illustre ce propos par un portage partiel de celle-ci utilisant Java à travers `O'Jacaré`. C'est aussi l'occasion de discuter du modèle de programmation fonctionnelle pour la construction d'interfaces graphiques.

[1] en utilisant l'utilitaire `lipo` qui porte bien son nom.

[2] `www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html`

Un autre vecteur de diffusion simplifié passe par l'exécution d'applications graphiques à l'intérieur d'un navigateur Web. En utilisant le *toplevel* embarqué d'Objective Caml on décrit le mécanisme de communication entre *applets* et navigateurs. Ces deux premières sections cherchent la portabilité des applications pour les utiliser sur les différentes plates-formes système/interface. La troisième section montre la construction d'applications Objective Caml spécifiques pour MacOSX et pour .NET. On conclut sur nos orientations futures pour une meilleure diffusion d'applications.

6.1 Interface graphique portable

La notion d'interface graphique portable ne date pas d'aujourd'hui ; un exemple sur lequel reposait CAML V2.6-1 est le «bitmap virtuel» de Le_Lisp. Objective Caml et son ancêtre Caml-Light proposent une bibliothèque simple, appelée `graphics`, qui offre une fenêtre graphique pour l'affichage de textes et le dessin ainsi qu'une gestion des événements. Une autre bibliothèque plus récente `camltk`, incluse depuis peu dans la distribution, donne des possibilités pour construire une interaction plus riche à base de *widgets*[3]. Elle repose sur le *toolkit* `Tk` défini initialement pour X-Window et lié au langage de commande `Tcl`. Objective Caml prend alors la place du langage de commande. Les commandes graphiques d'Objective Caml correspondent aux appels des commandes de `Tk`. Il n'y a pas d'intégration à la manière de Python et son utilisation nécessite de connaître la documentation de `Tk`. `Tk` est actuellement porté sur Windows et en cours de portage pour MacOSX. Il existe une troisième bibliothèque graphique accessible d'Objective Caml, appelée `labelGTK`[4], basée sur `Gtk` fournissant les types et les fonctions de liaison avec `Gtk` à la manière de `Tk`. Ces deux dernières bibliothèques utilisent pour la liaison avec Objective Caml des traits récents du langage comme les labels, les arguments optionnels et les variants. Là aussi la connaissance de la bibliothèque d'origine est nécessaire pour la construction d'une interface en Objective Caml. Cette difficulté de choix d'une bibliothèque graphique n'est pas directement liée au langage Objective Caml mais bien à la complexité croissante de l'interaction des programmes avec l'utilisateur. Le langage propriétaire Scol a connu pendant sa courte existence commerciale au moins trois versions incompatibles pour ses interfaces. Java a introduit dès sa première version l'API `AWT` qui a subi des changements importants pour la gestion des événements et est en partie doublée par la nouvelle bibliothèque `Swing` toujours proposée par SUN.

On rencontre en fait deux problèmes. Le premier est de s'interfacer correctement avec une bibliothèque existante et portable. C'est en règle générale atteint. La deuxième difficulté est d'intégrer dans le modèle du langage Objective Caml ces nouvelles fonctionnalités. Et là, la chose est moins bien réussie. En fait il n'est pas nécessaire de pouvoir utiliser toutes les fonctionnalités d'une bibliothèque graphique existante mais plutôt que l'on obtienne au final une bonne intégration dans son langage préféré.

On retrouve cette probable trop grande diversité pour d'autres langages fonctionnels comme par exemple Haskell[5] où l'on dénombre pas moins d'une dizaine d'interfaces graphiques dont des liaisons avec `Tk`, `Gtk` et `wxWindows`[6] (une interface portable sur les principaux systèmes de fenêtres) mais aussi des bibliothèques particulières plus fonctionnelles comme Fudgets ou Fran pour les animations.

Dans la mesure où les *GUI* suivent un modèle objet on peut se demander alors si le modèle fonctionnel est approprié pour effectuer ce genre de tâches. La construction d'une interface graphique peut être assimilée à la composition au sens fonctionnel des différents composants qui la constituent. Le traitement des événements peut être effectué par des fonctions réflexes (*callback*) liées au composant. Seule l'extension des composants graphiques existants nécessite une extensibilité correspondant davantage au modèle de programmation par objet.

Pour notre propos sur la portabilité, le choix de `graphics` semble le plus robuste même si ses possibilités sont restreintes.

---

[3] *window gadgets.*

[4] `wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lablgtk.html`

[5] `www.haskell.org/libraries`

[6] `www.wxwindows.org/`

6.1.1 Bibliothèque `graphics`

Bien que de bas niveau, `graphics` est probablement l'interface que l'on rencontre le plus fréquement dans les programmes Objective Caml. Elle est mono-fenêtre et autorise une interaction simple avec l'utilisateur. La figure 6.1 affiche la fin de partie du jeu `StoneHenge` de Reiner Kinzia[7] tiré de [41]. L'affichage est effectué sous X-Window.

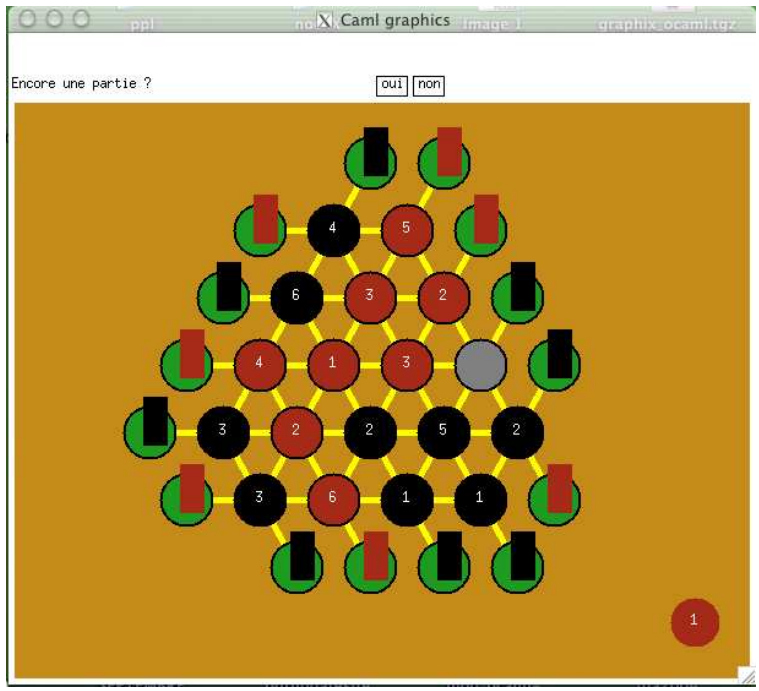


FIG. 6.1 – StoneHenge (X-window)

Chaque pièce ronde numérotée a été sélectionnée à la souris puis placée sur le plateau du jeu. La sélection d'une pièce a été réalisée par le programme qui détermine selon les coordonnées du clic souris si une pièce libre correspond.

Dans [41] nous avons essayé de construire au dessus de `graphics` une bibliothèque de plus haut niveau appelée `UPI`[8]. Voici sa description

> « *Cette interface graphique manipule des «composants». Un composant est une zone de la fenêtre principale qui peut être affichée, dans un certain contexte graphique, et traiter des événements qui lui sont destinés. Il y a principalement deux types de composants : les composants simples, comme un bouton de confirmation ou une zone de saisie d'un texte, et les «conteneurs» qui acceptent de recevoir dans leur zone d'autres composants. Un composant ne peut être attaché qu'à un seul conteneur. L'interface d'une application devient alors un arbre dont la racine correspond au conteneur principal (la fenêtre graphique), les nœuds sont d'autres conteneurs et les feuilles des composants simples ou des conteneurs sans descendant. Cette arborescence facilite la propagation des événements produits par l'interaction avec l'utilisateur. Si un composant reçoit un événement, il vérifie si l'un de ses descendants peut le traiter, si oui il le lui envoie et sinon il effectue l'action associée à cet événement.* »

Les applications construites au dessus d'UPI composent plus facilement l'interface et son interaction en utilisant la gestion des composants et des événements prédéfinis. La figure 6.2 montre l'exemple du convertisseur

[7] `www.knizia.de`
[8] Une Petite Interface.

francs-euros.

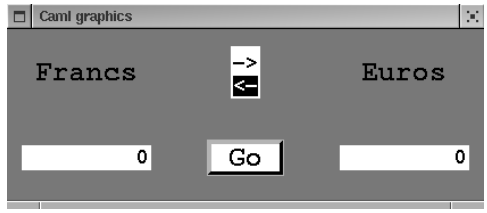


FIG. 6.2 – Convertisseur Francs-Euros (X-window)

Quatre types de composants apparaissent : deux labels (Francs et Euros), une liste de choix (`->`, `<-`), deux champs de texte et un bouton à cliquer. Sur cet exemple le programme correspond à l'interface graphique (environ deux pages de listing) et le calcul proprement dit à une ligne.

La programmation événementielle à la différence de la programmation fonctionnelle (qui compose les calculs) et de la programmation impérative (qui travaille en séquence) attend un événement pour le traiter. Ce traitement effectue une action en modifiant un état et en répercutant cette modification aux autres composants. L'expérience d'Upi a montré que les valeurs fonctionnelles permettaient facilement d'associer des réflexes aux composants. L'environnement d'une fermeture capture l'état à modifier et renvoie cette information par rapport aux composants dépendant de celui-ci. `graphics` est en même temps de trop bas niveau et de trop haut niveau pour la construction d'une *GUI*. L'impossibilité d'ouvrir plus d'une fenêtre est trop limitant et la gestion des événements est déjà figée.

Dans la construction d'une interface d'un programme on essaie de séparer l'interaction du calcul. Le modèle MVC issu du monde objet pousse à une séparation forte entre le modèle (qui contient l'état et effectue les calculs) de la vue qui affiche la partie visible de cet état du contrôleur qui gère les actions utilisateur. Cela permet de modifier aisément la partie visuelle et interactive sans modifier le code des calculs. Nous allons utiliser une version légère de ce modèle pour simuler `graphics` en Java dans le but d'interfacer la visionneuse `mlDvi` bâtie sur cette bibliothèque.

6.1.2 Une visionneuse `dvi` : jDvi

jDvi est une visionneuse `dvi`[9](voir figure 6.4) construite à partir de `mlDvi`[10] écrite en Objective Camli que l'on a modifiée pour l'interfacer avec Java grâce à `O'Jacaré`. Cette interface est conçue dans un schéma Modèle-Vue-Contrôleur. Les parties «Vue» et «Contrôleur» ont été séparées de manière explicite. L'objet `GrControler` contient une file d'attente d'événements, interrogée régulièrement par le «Modèle» et `GrView` est un `Canvas AWT`. Cette séparation permet notamment une fois le programme compilé, de modifier l'interface en ne touchant qu'aux fichiers Java et sans repasser par une phase d'édition de liens ; le programme engendré vérifie à l'initialisation la disponibilité des méthodes définies dans l'*IDL* (figure 6.3). L'interface `MlDvi`, déclarée callback, permet d'utiliser depuis Java un objet implanté entièrement en Objective Caml : c'est le point d'accès au «Modèle». Cet objet sera transmis à Java lors de l'appel au `main` Java. La classe Image, n'est utile que pour introduire un type.

L'application obtenue nécessite l'installation de LaTeX pour la construction des polices. Elle est néanmoins hautement portable. Bien que `graphics` ne soit pas porté nativement sous MacOSX, `jDvi` fonctionne pour ce système. On note même qu'elle est plus rapide que son équivalent sous X-Window sur MacOSX principalement pour un raffraîchissement des fenêtres moins systématique.

[9]DeVice Independent.
[10]`pauillac.inria.fr/~miquel/mldvi-1.0.tar.gz`

```
package java.awt;
class Image { }

package mypack;
[name ml_dvi, callback] interface MlDvi {
  void run(string, GrView, GrControler);
}

class GrView {
  static final int transp;
  int width;
  int height;
  void init(int,int);
  void clear();
  void close();
  void setColor(int);
  void fillRect(int, int,int, int);
  void drawImage(java.awt.Image,int, int);
  java.awt.Image makeImage(int[],int,int);
}
```

```
class DviFrame {
  static void main(MlDvi, string[]);
}

class CamlEvent {
  static final int KEY_PRESSED_MASK ;
  static final int BUTTON_DOWN_MASK ;
  static final int BUTTON_UP_MASK ;
  static final int MOUSE_MOTION_MASK ;

  final int mouse_x;
  final int mouse_y;
  final boolean button;
  final boolean keypressed;
  final char key;
}

interface GrControler {
  CamlEvent waitBlockingNextEvent(int);
  CamlEvent pollNextEvent(int);
}
```

FIG. 6.3 – Interface Vue et Contrôleur pour jDvi

6.2 *Plugin* Web pour des applications CAML

Une des motivations de l'embarquement du *toplevel* a été de construire un *plugin* d'exécution de programmes Objective Caml pour les navigateurs *web* qui soit multi-systèmes. Les navigateurs *web* sont des programmes largement utilisés dont l'interface, bien que pauvre, est bien acceptée. Il n'est pas déroutant pour un utilisateur *lambda* d'exécuter un programme comme une *applet* Java dans une fenêtre de son navigateur. Les fonctionnalités d'un navigateur peuvent être étendues pour traiter de nouveaux types de données ou de programmes grâce à un système d'extensions intégré ou *plugin*. Un *plugin* pour un langage devient alors un vecteur de diffusion simplifiée pour les applications écrites dans ce langage et contribue au succès de ce dernier.

En 1996, Francois Rouaix a développé un navigateur WWW, nommé MMM[130][11] et prononcé Meuh, pour la lecture de pages HTML et l'exécution d'*applets* CAML. Notre démarche est différente dans la mesure où l'on cherche à étendre des navigateurs existants grand public comme Netscape. L'idée est alors d'embarquer le *toplevel* Objective Caml dans un *plugin* Netscape [110]. Le programme Objective Caml téléchargé sera transmis au *toplevel* qui exécutera l'application graphique dans la fenêtre du navigateur.

Cette idée avait été expérimenté lors d'un TER[12][118] en lançant pour chaque programme graphique Objective Caml (utilisant la bibliothèque `graphics`) téléchargé un processus *toplevel* qui communiquait avec la fenêtre du navigateur. Cela fonctionnait sous X-Window dans la mesure où une application peut utiliser une fenêtre graphique d'une autre application si elle connaît l'identificateur de la fenêtre concernée. Mais pour d'autres systèmes de fenêtrage comme Windows ou MacOSX cela n'est pas possible. En revanche un *toplevel* embarqué dans un *plugin* pourra accéder et dessiner dans une fenêtre du navigateur.

Pour apprécier cette argumentation, on décrit tout d'abord le mécanisme de base de l'exécution d'un *plugin* pour ensuite montrer comment le *toplevel* peut interagir via la bibliothèque `graphics` avec une fenêtre d'un navigateur. On illustre cette réalisation par des applications existantes n'ayant pas eu besoin d'être modifiées.

[11] `pauillac.inria.fr/~rouaix/mmm`
[12] Travaux Encadrés de Recherche.

FIG. 6.4 – Capture d'écran de `jDvi`

On discute ensuite des choix d'implantation et des limitations de cette première intégration.

6.2.1 Développement d'un *plugin* pour Netscape

Le navigateur Netscape fournit une bibliothèque de développement [110] pour la réalisation de *plugins*. De nombreux autres navigateurs, comme Mozilla, Konqueror ou Safari, utilisent cette approche à l'exception récente et notable d'Internet Explorer.

Un *plugin* peut dessiner dans une fenêtre à part ou dans une partie de la fenêtre du navigateur. On s'intéresse à ce dernier cas.

On indique la présence d'un plugin dans une page HTML par une balise `EMBED` (ou `OBJECT`). Le source suivant indique d'utiliser un plugin traitant du type `x-caml-plugin` dans la page du navigateur dans une fenêtre de taille 400x300 en lui envoyant le flux correspondant au fichier `colwheel.ml` :

colwheel.html
```
<EMBED type=application/x-caml-plugin
  src="http://www.pps.jussieu.fr/~emmanuel/colwheel.ml" width=400 height=300>
```

Modèle d'exécution d'un *plugin* Au démarrage d'un navigateur, celui-ci liste les *plugins* disponibles et les enregistre en fonction des types MIME qu'ils traitent. Quand le navigateur rencontre des données d'un type MIME enregistré par un *plugin*, il charge dynamiquement le plugin en mémoire (s'il n'est pas déjà chargé) et crée une nouvelle instance du *plugin*. Il peut avoir plusieurs instances d'un *plugin* à un moment donné (plusieurs balises EMBED dans une même page ou plusieurs pages avec une telle balise). Quand l'utilisateur quitte la page ou détruit la fenêtre l'instance du *plugin* est détruite. Quand toutes les instances d'un plugin sont détruites, celui-ci est libéré de la mémoire.

Lors de la création d'une instance d'un *plugin* le navigateur lui indique le descripteur de la fenêtre graphique et si besoin est crée un flot de communication pour transmettre les données à traiter. Le *plugin* doit pouvoir répondre aux appels de fonction provenant du navigateur. Ce dernier peut être aussi appelé par le *plugin*. L'*API* de développement fournit l'ensemble de ces fonctions.

API de développement Il y a trois types de fonctions présentes dans la bibliothèque de développement de Netscape :

1. les fonctions de chargement et de libération du *plugin* (préfixe `NP_`) ;
2. les fonctions d'interface du *plugin* qui seront appelées par le navigateur (préfixe `NPP_`) ; à écrire par le programmeur ;

3. les fonctions d'interface du navigateur qui seront appelées par le *plugin* (préfixe `NPN_`).

Lors de la détection d'une page contenant des données à traiter par un *plugin*, le navigateur effectue les actions de la figure 6.5 :

- il charge dynamiquement en mémoire le code du *plugin* (s'il ne l'est pas déjà) par la fonction `NP_Initialize` ;
- il crée une nouvelle instance du *plugin* (fonction `NPP_New`) sachant que plusieurs instances peuvent être créées si nécessaire ;
- il informe le *plugin* de l'identificateur de la fenêtre (`NPP_SetWindow`) ;
- il transmet les données aux *plugin* via un flux de communication et les événements survenant dans la fenêtre du *plugin* ;
- quand la page de l'instance disparaît l'instance est détruite (fonction `NPP_Destroy`) ;
- quand la dernière instance d'un *plugin* est détruite alors le *plugin* est déchargé de la mémoire (fonction `NP_Shutdown`).

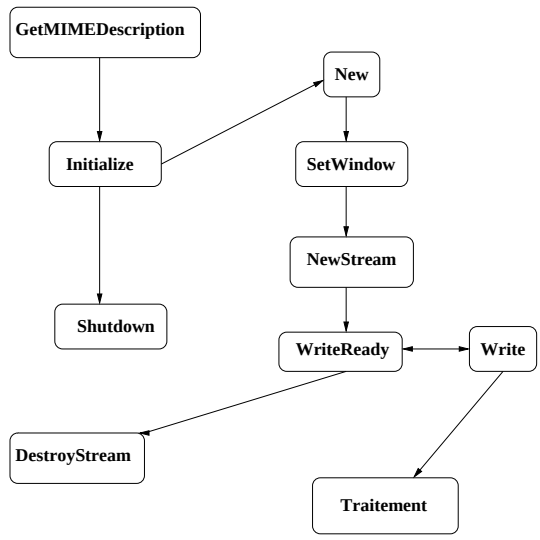


FIG. 6.5 – Schéma d'exécution d'un *plugin*

Le *plugin* peut alors gérer les événements de la fenêtre qui lui est associée par le navigateur en définissant ses propres fonctions de traitement d'événements et d'affichage.

6.2.2 *Plugin* pour les programmes Objective Caml

Le *plugin* pour les applications graphiques Objective Caml suit la démarche précédente. Il est fourni sous forme d'une bibliothèque dynamique (`.so` pour Unix, `.dll` pour Windows). Lors de la demande de création de la première instance `NPP_New` l'identificateur de la fenêtre graphique fournie par le navigateur est conservée. À l'envoi des données dans le flux de communication un *toplevel*, construit sous forme d'une bibliothèque dynamique, est lancé. Celui-ci charge la bibliothèque `graphics`, met à jour la fenêtre graphique puis charge les données passées dans le *toplevel*. La bibliothèque `graphics` a été modifiée pour pouvoir forcer l'identificateur de la fenêtre (`set_window_id`).

```
  win = ((PluginInstance *) instance->pdata)->window;

  sprintf(headers,"Graphics.set_window_id %lu ;;",win);
  sprintf(src,"#use %s;;",fname);
  commandes[0]=headers;
  commandes[1]=src;
  pthread_create(&thread, NULL, (void*)toplevel_thread, (void*)commandes);

void * toplevel_thread(char* arg[2]) { ...
 (*toplevel_exec)(arg[0]);
 (*toplevel_exec)(arg[1]);  ...
}
```

Comme plusieurs instances peuvent être lancées et que la bibliothèque `graphics` ne peut adresser qu'une

seule fenêtre graphique, il ne peut pas y avoir plusieurs programmes Objective Caml s'exécutant dans le même navigateur. Pour éviter de bloquer le navigateur par l'exécution du programme Objective Caml le lancement d'un programme est *threadé*. Néanmoins un seul *thread* est actif à un moment donné.

Les figures 6.6 et 6.7 montrent respectivement les applications graphiques `colorwheel`[13] dans le navigateur Mozilla sous X-Window et `deminer`[41] dans le navigteur FireBird sous Windows.

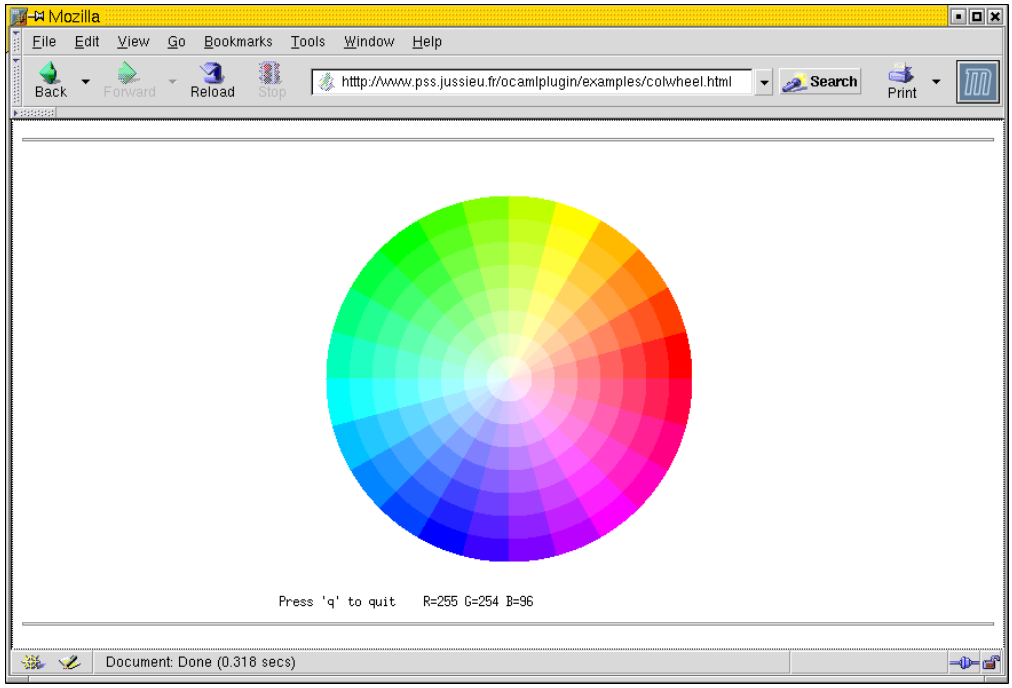


FIG. 6.6 – L'application Colorwheel sous Mozilla (X-window)

La sortie de `Colorwheel` (appui de la touche 'q') efface la portion de fenêtre graphique du navigateur associée à l'application Objective Caml..

6.2.3 Choix d'implantation et limitations

Le choix de la bibliothèque `graphics` a été effectué pour sa portabilité. Cela oblige à n'avoir qu'une seule application graphique s'exécutant en même temps parce que cette bibliothèque ne sait gérer qu'une seule fenêtre. Ce *plugin* est en fait mono-*thread*.

Du point de vue sécurité des programmes téléchargés, la compilation des sources garantit que le code-octet exécuté est correct (dans la mesure où la bibliothèque *toplevel*) l'est. Rien n'empêche par ailleurs de charger du code-octet. Les temps de transfert et compilation d'un texte source sont le plus souvent équivalents aux temps de transfert et de chargement du code-octet. Toutes les fonctions de la bibliothèque standard sont accessibles, y compris les lectures/écritures de fichiers. Une manière d'y remédier est de suivre les limitations de la `stdlib` comme présentées pour MMM.

L'application graphique doit être contenue dans un seul fichier source. Une évolution est de pouvoir télécharger des fichiers d'un serveur *via* la primitive `use` en ayant accès à l'URL du serveur.

[13] `caml.inria.fr/Examples/eng.html`

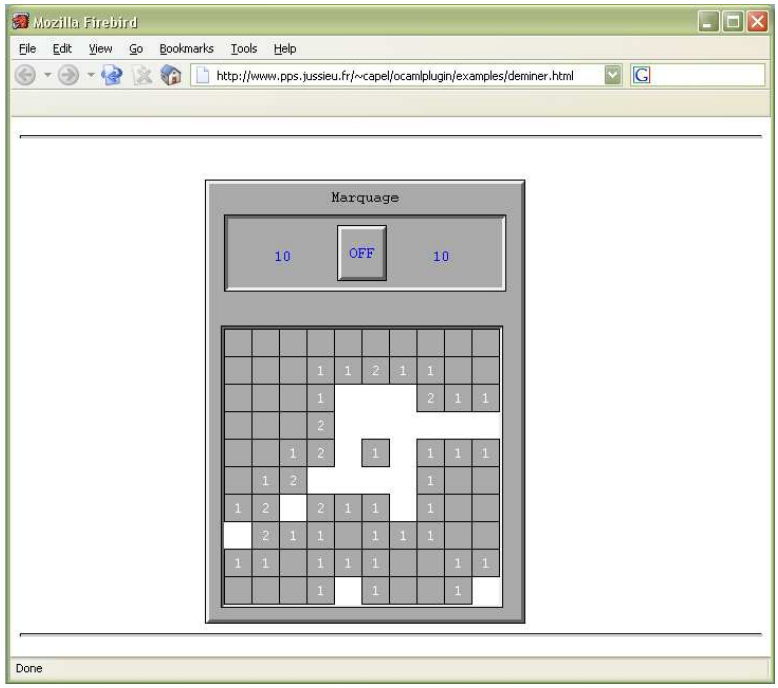


FIG. 6.7 – L'application Deminer sous FireBird (Windows)

6.3 Systèmes spécifiques

Comme bien souvent l'on désire profiter de l'ensemble des possibilités offertes par un système d'exploitation et son système de gestion de fenêtres tout en étant portable. Nous nous intéressons ici aux particularités de diffusion et de régionalisation des systèmes MacOSX et Windows pour en tirer des enseignements pour des solutions portables.

6.3.1 Applications MacOSX-Cocoa

Les applications développées sous MacOSX ont été créées le plus souvent avec les outils Project Builder associé à Interface Builder issus du monde NeXT. Elles sont écrites en Objective C, extension objet à la SmallTalk de C. Une application contient des fichiers de ressource, des déclarations de liste de propriétés et le code exécutable. Cet ensemble de fichiers est regroupé dans un catalogue et des sous-catalogues. Les ressources contiennent en particulier l'ensemble des textes de l'interface. Ceux-ci peuvent alors être adaptés à une autre langue pour une diffusion multi-langues. Il n'est pas nécessaire de recompiler l'application. Le choix de la langue détermine l'emplacement de la ressource à employer. Les chaînes de caractères des textes sont stockés sous forme d'objets Objective C dans des fichiers de langues. Au lancement d'une application ces objets sont chargés dans l'application qui peut alors les afficher.

Il y a deux possibilités pour construire des applications écrites en Objective Caml pour MacOSX. La première est d'inclure son programme Objective Caml dans une interface construite en Objective C ; le code Objective Caml étant alors associé à des actions utilisateur. Jérôme Vouillon avait de cette manière construit un environnement de programmation Caml-Light sous NextStep où l'on pouvait explorer les modules simples, exécuter du code dans le *toplevel* et explorer sous forme d'arbre les valeurs ML. Pour cela, il avait réalisé une moulinette d'interfaçage entre l'API `AppKit` et Caml-Light qui définissait à partir des fichiers d'interface `.h` les définitions `external` de Caml-Light quand cela était possible et les ignorait sinon. La deuxième possibilité est de porter sous MacOSX une bibliothèque graphique utilisée pour Objective Caml comme `graphics` et de

définir un constructeur d'application spécifique. Ce travail est en cours de finition avec Grégoire Henry. La figure 6.8 montre une capture de comparaison de tris[14].

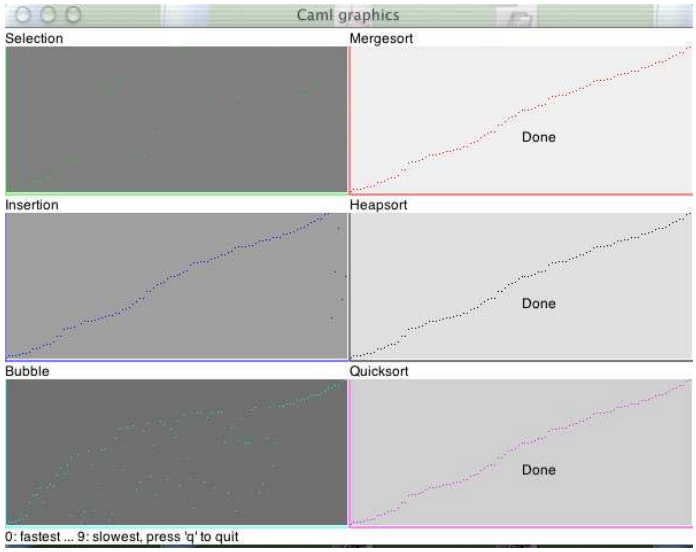


FIG. 6.8 – Sort (Cocoa)

Cette application peut être lancée par simple clic et possède son menu propre aux applications MacOSX. Parce que l'interaction peut venir de la fenêtre de `graphics` et de la barre des menus, l'application Objective Caml une fois compilée en C (voir chapitre 3) est lancée dans un *thread* (`CamlThread` par le délégué principal de l'application).

main.m

```
#import <Foundation/Foundation.h>
#import <AppKit/AppKit.h>

#import "StartupDelegate.h"

int main (int argc, const char * argv[])
{
  NSAutoreleasePool * pool;
  pool = [[NSAutoreleasePool alloc] init];

  [NSApplication sharedApplication];
  [NSApp setDelegate: [[StartupDelegate alloc] init]];

  [NSApp run];

  [pool release];
  return 0;
}
```

CamlThread.m

```
#import "CamlThread.h"
#include <caml/mlvalues.h>
#include <caml/callback.h>

@implementation CamlThread
-(void)run {
  NSAutoreleasePool * pool;
  pool = [[NSAutoreleasePool alloc] init];
  NSLog(@"Début  du thread Caml");

  char * argv[0] = ;
  caml_startup(argv);

  NSLog(@"Fin du thread Caml");
  [NSApp terminate:nil];
  [pool release];
  return;
}
@end
```

La commande (écrite en Objective Caml) de fabrication d'applications effectue ces tâches automatiquement et construit les différents fichiers de ressource. L'application obtenue peut alors être diffusée de manière autonome de la distribution d'Objective Caml ce qui était le but recherché.

Par contre comme les différents messages d'affichage sont définis dans le source du programme Objective Caml, seule la barre de menus peut être régionalisée. Il est donc nécessaire d'extraire du programme lui-même

[14] `http://caml.inria.fr/Examples/eng.html`

les informations textuelles spécifiques à une langue. Cela dépasse le cadre des applications MacOSX et nous y reviendrons dans la dernière section de ce chapitre.

6.3.2 Applications Windows

Les applications Objective Caml sont rarement distribuées pour Windows. On cherche ici à vérifier les facilités de distribution et les implications de la régionalisation offertes par .NET.

tri chinois La figure 6.9 illustre les possibilités de manipulation offertes par le codage *unicode* utilisé ici pour le chinois. Pour visualiser ces caractères on utilise une version expérimentale du *toplevel* de O'CAMiL. L'interfaçage avec les bibliothèques .NET utilise la couche de bas niveau `external` pour appeler une méthode statique.

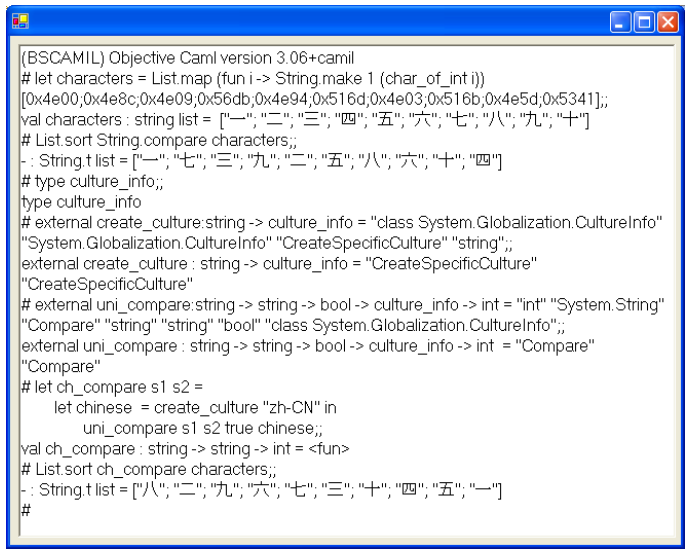


FIG. 6.9 – Comparaison dépendante des spécificités de culture.

La liste `characters` contient les codes unicode des nombre chinois de 1 à 10 (*yi* , *er*, *san*, *si*, *wu*, *liu*, *qi*, *ba*, *jiu*, *shi* en Pinyin[15]). Le premier tri utilise la fonction standard de comparaison des chaînes (`String.compare`) qui utilise l'ordre des codes. La classe `System.Globalization` de .NET fournit des outils qui reflètent les spécificités culturelles, ce qui permet alors de trier en respectant l'ordre Pinyin.

6.4 Conclusion : pour une meilleure diffusion

En dehors du cercle restreint des concepteurs, un langage est apprécié principalement par ses applications développées. La facilité d'installation et le confort d'utilisation de celles-ci font partie des critères importants pour cette appréciation. Nous avons montré dans ce chapitre les besoins de portabilité et de spécificité dans l'intégration des applications tant au niveau du système d'exploitation que du système de gestion de fenêtre associé. Les expérimentations présentées sont directement issues des travaux sur l'interopérabilité développés au chapitre 3. Elles ont permis de montrer l'importance de l'interface graphique pour la diffusion d'applications tant pour faciliter l'utilisation des programmes que comme frein à leur portage.

L'exemple de la visionneuse `jDvi` utilisant l'AWT de Java *via* `O'Jacaré` et simulant `graphics` a permis de construire une application portable y compris sur MacOSX où `graphics` n'est pas portée. L'embarquement

[15] La transcription officielle du Mandarin.

du *toplevel* dans des navigateurs Web utilise le mécanisme de *plugin* pour étendre les fonctionnalités d'une application existante. La construction d'applications graphiques autonomes pour MacOSX ou l'encapsulation d'`hevea` comme application autonome .NET permettent une diffusion plus large.

On obtient alors soit des applications portables utilisant `graphics`, soit des applications embarquées avec un *GUI* particulier (Forms, AWT, AppKit). Comme `graphics` est maintenant trop limitée, la question se pose d'utiliser pour CAML une véritable *GUI*. Doit-on alors en construire une nouvelle à partir de bibliothèques portables (`Tk`, `Gtk`, `wxWindows`, ...) ou à partir de celles d'un langage ou d'un environnement portable (Java, .NET), en les intégrant et probablement en les épurant ? Ou bien utiliser des *GUI* existantes dans d'autres langages en spécifiant les interactions possibles entre celles-ci et la partie calculatoire d'un programme à la manière du modèle MVC ? La réponse n'est pas immédiate et dépend des outils de construction d'interface. Il est à noter que l'expression dans un langage de programmation d'une interface utilisateur est très verbeuse et contient du code lourd à gérer manuellement. Celui-ci pollue en général la lisibilité d'un programme d'où l'intérêt d'une bonne séparation.

Cette nécessaire séparation se retrouve pour la régionalisation des applications. Si la régionalisation s'effectue uniquement au niveau de l'interface utilisateur, on retombe sur les questions précédentes et sur les besoins d'outils pour sa gestion. La possibilité d'étendre une application à une langue donnée doit pouvoir s'effectuer sans recompilation de la dite application. Néanmoins cette information doit être répercutée au sein même de l'application non seulement pour les dialogues avec l'utilisateur mais aussi dans le calcul si besoin est.

Autant le modèle fonctionnel typé est approprié pour la construction d'interface (composition de composants et traitement d'événements par fonctions réflexes) autant il ne se prête guère à l'extensibilité des composants. Le modèle objet de par sa liaison retardée peut répondre à ce besoin. Mais ce n'est pas forcément le seul modèle. L'interface d'une application peut aussi être décrite dans un langage de description externe à la manière d'UIML[16]. Bien souvent c'est l'outil de développement qui rend la construction de l'interface praticable.

L'installation d'une application Objective Caml pour Windows ou MaOSX est facilitée respectivement par la production d'applications .NET de O'CAMiL et par l'encapsulation en Objective C d'une application Objective Caml. Leurs mises à jour sont liées aux bibliothèques dynamiques. Dans le cas de .NET la gestion de version autorise la cohabitation de différentes versions d'une même bibliothèque. L'expérimentation du *plugin* pour navigateur Web montre une troisième voie de diffusion par l'envoi du source qui est recompilé à la volée. Ces sources peuvent être cryptées si besoin est.

[16] www.uiml.org/index.php

Conclusion

On mesure aisément les phénomènes de mode dans l'industrie informatique par les stages industriels proposés à nos étudiants. Après les besoins de conception et de portage d'application sous interface graphique est venue la vague de la programmation objet qui a pris son essor au moment de l'apparition de Java promouvant la programmation client-serveur via les navigateurs web. On atteint maintenant la programmation par composants tout en ajoutant des couches de protocoles et de manipulations des données sous format se voulant universel à la manière de XML, sans oublier la répartition par les *web services*.

On peut se demander avec raison où est la place des langages fonctionnels statiquement typés dans ces nouveautés renouvelées chaque année. Et pourtant ce côté novateur principalement poussé par le marketing reprend bien souvent des solutions anciennes. Il s'écoule environ trente ans entre le premier langage à objets (SIMULA) et l'arrivée massive de Java. La présentation du langage Java par Christian Queinnec «Un cocktail innovant de solutions éculées» décrit bien la situation.

Au delà du langage ML quelles sont les influences que la programmation fonctionnelle typée peut apporter aux évolutions de la science (ou technologie) informatique ? Pour y répondre nous précisons tout d'abord notre réponse aux critiques présentées dans l'introduction pour détailler ensuite l'importance du typage statique pour cette famille de langages et quels sont ses éléments essentiels. Nous essayons ensuite de dégager quelle sorte de cohabitation peut être enrichissante entre les déferlantes Java,C#,VB.NET et les langages fonctionnels statiquement typés. Ce sont ces réponses qui orientent nos travaux futurs.

Réponses aux critiques

La présentation de ce travail voulait répondre aux critiques anciennes et récentes sur les langages fonctionnels. Elle montrait de par la diversité des thèmes abordés la vitalité de la recherche sur les langages. Celle-ci n'est pas cloisonnée pour un type de langage car les travaux sur les différentes familles de langages s'influencent.

Les compilateurs CeML et O'CAMiL ont montré que l'on pouvait construire du code portable. Dans les années 90, C semblait avoir atteint le statut d'asssembleur portable. C'est dans cet esprit que CeML avait été conçu et il a ainsi profité des améliorations réalisées pour les compilateurs C durant la décennie. Il montre ainsi que portabilité et efficacité peuvent faire bon ménage. Le retour des machines virtuelles (JVM, .NET) remet en cause en partie ce statut. Certes ces machines sont conçues pour faciliter la compilation des langages à objets mais elles ou leurs successeurs peuvent devenir le creuset d'applications multi-langages. On montre avec O'CAMiL que les langages fonctionnels peuvent faire partie de l'offre «standard» pour le programmeur. On marie ici les possibilités offertes par la portabilité et l'interopérabilité. Certes l'efficacité de O'CAMiL n'est pas encore au rendez-vous pour les programmes très fonctionnels, mais outre le fait qu'il est encore jeune, la compilation vers des machines virtuelles typées pour langages à objet nécessite de reconsidérer certaines habitudes de la compilation des langages fonctionnels statiquement typés, en particulier sur la conservation de l'information de typage tout au long du processus de compilation.

Les travaux sur l'interopérabilité ont grandement avancé sur ces dix dernières années. Nous avons montré

qu'une application pouvait être écrite dans plusieurs langages comme par exemple `jDvi` en Objective Caml et Java passant par C. La tendance amorcée avec Java et reprise par .NET nous semble en théorie ouverte aux langages fonctionnels typés.

Les applications réalisées en Objective Caml deviennent de plus en plus complexes et utilisent des bibliothèques spécifiques ou des commandes externes. Nous nous sommes aperçus au moment de tester O'CAMiL en compilant les applications phares d'Objective Caml qu'elles reposaient sur des composants externes `:Unison` avec `Tk`, *threads* et primitives réseau, `mlDvi` par l'installation de certains outils de LaTeX, `Active-Dvi` reposait sur des mécanismes de base de X-Window pour l'animation des présentations. Seul le compilateur et certains assistants de preuves étaient directement portables en O'CAMiL. C'est un phénomène compréhensible parce que les applications actuelles, liées au multimédia effectuent des tâches de plus en plus complexes. Un travail de définition de certaines bibliothèques, principalement pour l'interface graphique est nécessaire aujourd'hui. La phase d'expérimentation ayant eu lieu.

L'enseignement des langages fonctionnels reste important dans le milieu académique français dans les différents cycles universitaires et les classes préparatoires. Et c'est une bonne chose. Certes on peut entendre un certain nombre de critiques sur la difficulté d'apprentissage de ces langages, mais intrinsèquement le modèle fonctionnel est plus simple que le modèle objet, d'où peut-être une nécessité d'adaptation des cours comme indiqué à la conclusion du chapitre 5. Un cours de programmation n'a pas à commencer par un cours de logique, même si celui-ci peut aider à la formalisation. Il existe de nombreux polycopiés adoptant cette démarche plus pragmatique. Il manque pour Objective Caml un environnement simple de prise en main à la manière de DrScheme[17]. Il faut noter que l'enseignement des langages fonctionnels est un grand plaisir pour l'enseignant que je suis et pour les étudiants quand ils ont franchi la première marche de la compréhension du modèle.

Le chapitre 4 a montré deux pistes pour la construction de véritables environnements de développement (intégrés ou non). C'est encore actuellement un des points faibles. L'expérience de `CamlNext` était à l'époque une avancée remarquable. On peut noter qu'une bonne interopérabilité avec des outils existants peut aider à leurs réalisations. Néanmoins la vision classique du typage comme sûreté de fonctionnement doit être ouverte pour que l'information de types continue d'exister à l'exécution ou à la mise au point. Ces travaux ne sont pas fortement promus comme cadre de recherche. C'est un tort.

La standardisation des langages fonctionnels a connu principalement une vraie norme pour Scheme et une formalisation pour Standard ML. Néanmoins on a vu surgir pour ces deux langages des variantes comme Guile pour Scheme ou SML.NET pour SML qui introduisaient de nouveaux traits. Objective Caml est resté laboratoire expérimental en évoluant au fil des ans mais en essayant de conserver la compatibilité ascendante. Cela a fonctionné dans la mesure où il y avait une source principale (le groupe Cristal de l'Inria) de décision dans son évolution. Néanmoins cela a pu freiner son adoption dans le milieu industriel. On peut noter qu'un langage comme Java évolue aussi de manière directive de la part de SUN. Néanmoins on peut imaginer qu'un langage a besoin de points de stabilité à la manière d'Ada (ADA83 et ADA95), quitte à en diverger entre deux périodes. De toute manière il existe une certaine anarchie et un standard n'est pas forcément gage de pérennité. L'établissement de consortium[18] peut permettre une évolution concertée.

La disponibilité des compilateurs des langages fonctionnels s'est grandement améliorée. Un exemple est le compilateur expérimental Bigloo pour .NET qui s'installe aisément sous Windows. L'ensemble des distributions existe en format binaire pour les principaux systèmes. Le point à améliorer concerne la diffusion d'applications et leurs mises à jour.

[17] `www.drscheme.org`
[18] `caml.inria.fr/consortium/index-fr.shtml`

Comme indiqué dans l'introduction, la saga de la commercialisation d'environnements pour la programmation fonctionnelle ne force pas l'optimisme. Même en quittant la communauté des développeurs comme a fait la société Cryo-networks en promouvant un langage fonctionnel pour les applications de communauté virtuelle a subit de plein fouet l'effondrement de la bulle Internet. Le fait que la majorité des implantations de langages fonctionnels soit en «open source» n'est pas forcément un frein à la constitution d'un marché. L'exemple de la société ACT[19] montre que le modèle est viable s'il existe une base d'applications suffisante. Ce type de besoin pour des langages comme Objective Caml ou Haskell pourra survenir par la diffusion d'applications «phares» motivant leur emploi. Ce discours pouvait déjà être tenu il y a quelques années, mais il semble qu'il reste valable.

de l'importance du typage Au long des différents chapitres de ce document nous avons rencontré des problématiques liées au polymorphisme paramétrique qui donne un certain niveau d'homogénéité à ML.

- **chapitre 1** : La compilation d'un langage statiquement typé se doit de conserver toute l'information de types (vérifiée ou inférée) durant la chaîne de transformation du texte du programme. Cela devient même crucial quand la cible du compilateur est un assembleur typé.
- **chapitre 2** : Les outils d'extensions syntaxiques gagneraient à manipuler le système de types.
- **chapitre 3** : L'interopérabilité entre langages nécessite le passage d'un système de types à un autre quitte à effectuer cette passerelle dans l'intersection des systèmes de types.
- **chapitre 4** : Un environnement de développement a besoin de communiquer au concepteur d'application l'information de types. Cela tant au niveau de l'écriture des interfaces et du programmes que pour la trace et la mise au point de programmes.
- **chapitre 5** : Un langage typé statiquement oriente la pensée de l'étudiant. On passe du «program first, think later» à «type first, think later». Sans être absolument révolutionnaire, c'est déjà un progrès.
- **chapitre 6** : Le typage statique peut être en partie vérifié, de manière dynamique, pendant la phase d'élaboration au chargement d'une application. La sûreté du point de vue typage du téléchargement d'applications peut être simplifié par la recompilation de sources.

éléments de ML

inférence et vérification de types Le typage statique est l'élément essentiel du langage ML. Il garantit que les erreurs de typage détectées dès la compilation ne surviendront pas à l'exécution. Même si un mécanisme comme les exceptions reste nécessaire dans la mesure où une division par zéro devra rompre le calcul en cours qui n'aboutit pas. Enlever à ML son typage statique c'est revenir dans le monde Lisp (excellent laboratoire libertaire au demeurant). Les langages typé dynamiquement considèrent une erreur de typage comme pouvant survenir et la traitent par des exceptions.

Supposer que l'ensemble d'un programme a passé l'épreuve de l'algorithme de typage de ML, c'est se limiter à un monde certes riche mais assez isolé. Cela survient y compris entre programmes issus du même langage désirant communiquer des valeurs complexes. Cela limite fortement l'interopérabilité entre langages. L'idée alors est d'ouvrir le carcan du typage statique par un typage dynamique dans les cas où les valeurs ou les programmes sont extérieurs. Pour rester dans l'esprit ML, ce typage dynamique doit rester explicite.

Mais il y a un point qui est souvent polémique dans le typage statique à la ML. Il est très souvent confondu avec l'inférence de types. Or ce qui est important c'est la vérification statique des types d'un programme. Ceux-ci peuvent bien sûr être fournis par le programmeur. Même si l'algorithme de vérification ressemble dans le cas du polymorphisme paramétrique à l'algorithme d'inférence. Il n'est pas choquant pour un programmeur Ada d'indiquer les types de paramètres d'une fonction ou les types d'un module générique.

Il faut alors considérer l'inférence de type comme un confort. Mais ce confort peut être trompeur pour le débutant et même le programmeur confirmé. Il n'est pas rare de voir plancher un étudiant sur une erreur de

[19] Ada Core Technologies : `www.gnat.com`

typage inférée par le compilateur. De plus le fait de ne pas à avoir à écrire les types influence le système de types du langage lui-même. La syntaxe des types peut alors devenir complexe, seules les définitions de types seront à la charge du programmeur.

filtrage de motifs La définition d'une fonction par cas sur un type somme est toujours élégant et d'une grande clarté. De plus les avertissements du filtrage que peut prodiguer un compilateur donne une information facilement compréhensible par le programmeur.

application partielle L'application partielle d'une fonction à une partie de ses arguments permet de ne pas se soucier de l'arité de celle-ci ou de sa définition curryfiée ou non. Néanmoins elle a un coût pour le code produit qui doit dans tous les cas en tenir compte. Cela est particulièrement vrai dans le cadre de polymorphisme paramétrique à la ML. Dans le modèle mixte fonctionnel/impératif il est nécessaire d'ajouter des *warnings* pour les évaluations d'expressions dans la séquence au cas où le programmeur aurait fait une erreur. Cela n'est pas entièrement satisfaisant. Par exemple Scheme ne l'intègre pas. Cela ne l'empêche pas de manipuler des fermetures.

modules et objets La structuration d'une application Objective Caml peut utiliser les modules paramétrés (foncteurs) ou définir une hiérarchie de classes, et bien sûr mixer les deux organisations. Les modules paramétrés dans le cadre des types abstraits de données souffrent du besoin de transmettre une information de types pour indiquer le partage entre les modules paramètres d'un foncteur. On revient à la discussion précédente entre inférence et vérification statique, et donc à la syntaxe des types et des signatures de tels modules. Le modèle objet d'Objective Caml permet d'une part de programmer dans un style fonctionnel/objet et d'autre part facilite l'interopérabilité avec d'autres langages objet sans avoir à modifier le langage. Certes son polymorphisme de rangées diffère du modèle objet habituel, mais peut pour une première lecture être présenté à la manière du polymorphisme d'inclusion des langages objets classiques. Il lui manque une résolution de la surcharge qui ne supporte pas les ambiguïtés liées à l'inférence de types. On s'aperçoit que la structuration d'applications a besoin d'information de types données par le programmeur.

Une cohabitation annoncée

En 1996 Christian Queinnec et Pierre Weis donnaient un état des lieux et perspectives pour la programmation applicative dans la revue TSI[125]. On y lisait :

> « *L'indéniable succès pédagogique actuel des langages applicatifs devrait s'accompagner dans les cinq à dix ans d'une percée de ces langages dans le monde industriel. Cette diffusion ne sera pas sans rappeler celle de Pascal dans les années 80, à ceci près que les langages applicatifs procurent des traits plus puissants qui leur conféreront sans doute une plus grande longévité.* »

Le colloque scientifique en l'honneur de Jean-François Perrot [20] sur le thème «20 ans après : où en sont les objets» s'est conclu par une table ronde commentant la récente discussion entre Richard Gabriel[64] et Guy Steele[145] sur «les objets ont-ils échoué ?». Christian Queinnec introduisait sa réponse par la phrase suivante :

> «*La question ne se pose absolument plus : les objets ont gagné... leur place dans les cerveaux des développeurs et dans les cœurs de leurs hiérarchies.*»

Que s'est-il passé pendant ces sept ans ? La vague Java ? Est-ce que les langages fonctionnels sont en voie de «Prologisation» ?

Pour répondre à ces questions liées, un regard sur le proche passé peut éclairer la situation actuelle. On peut séparer le développement des langages objets en deux branches, la branche «molle» de SmallTalk issue de la communauté Lisp et une branche plus «dure» comme Eiffel issue de l'ingénierie logicielle. Java apporte

une sorte de réconciliation, autorisant d'un côté la réflexion et le chargement dynamique tout en insistant sur l'importance du typage statique. Il réhabilite au passage la compilation vers des machines virtuelles, la gestion automatique de la mémoire et fait accepter le traçage des exceptions par le programmeur. Ce «cocktail innovant de solutions éculées» apparu au bon moment, soutenu par l'industrie, va se développer de façon incroyable (et difficilement prévisible à l'époque) de telle sorte qu'il ne reste quasiment que lui. Alors *exit* SmallTalk, Eiffel, Lisp ou ML ?

Il est peut-être préférable d'avoir un regard plus optimiste sur cette période. La conception et l'implantation de Java est en fait une chance pour la branche fonctionnelle parce qu'elle justifie *a posteriori* les choix d'implantation et de typage. Les présentations des environnements (machine abstraite, bibliothèque d'exécution, compilateur) sont quasiment identiques entre Java et Objective Caml. Certes il y a une influence du modèle objet y compris en CAML qui bâtit alors une extension objet[128], mais on peut aussi mesurer une influence inverse. Bien que le concepteur de Java insiste sur la sûreté apportée par le typage statique, les programmes Java nécessitent de nombreux tests de type dynamiques. Cela est principalement dû à l'absence de classes paramétrées (ou *templates* pour reprendre la terminologie C++). La prochaine version de Java intégrera un mécanisme proche de Generic Java[26] autorisant de telles déclarations et l'équipe de Microsoft à Cambridge travaille sur les *generics*[92] pour .NET. Ces nouvelles extensions se rapprochent alors du polymorphisme paramétrique à la ML.

Ces développements effectués en parallèle entre les mondes académiques pour les langages fonctionnels et les mondes industriels pour le modèle objet à la Java s'influencent. Il n'y a pas de raison que cela change dans le futur proche. On voit difficilement la communauté académique des langages fonctionnels typés passer aux objets tels que ceux de Java. De l'autre côté l'attrait du fonctionnel typé dans l'industrie ne risque pas de subir plus qu'un frémissement dans le cas où des applications «phares» arrivent à promouvoir le modèle.

Néanmoins il est intéressant de déterminer dans quels domaines les influences vont s'effectuer. Le «durcissement» des langages objets apportés par les *generics* va autoriser certaines analyses statiques des programmes se rapprochant de celles effectuées en ML comme l'analyse de flots d'informations[120], de détection d'échappement d'exceptions ou sur les systèmes de types objets, comme ML-sub [25], pouvant déboucher sur des prototypes pour Java comme Nice[20]. L'évolution de la structuration peut aussi rejoindre des préoccupations issues des objets comme les *mixin*[84].

De l'autre côté, le foisonnement d'environnements, la richesse des bibliothèques des outils industriels ainsi que les méthodologies de développement, ne peuvent laisser insensible le programmeur. Les applications informatiques s'attaquent à des problèmes de plus en plus nombreux et de plus en plus complexes. Il devient alors nécessaire de pouvoir utiliser de tels outils ou du moins d'interopérer avec eux. Le programmeur *lambda* a pris l'habitude d'un certain confort de travail de par l'utilisation de ces outils et environnements. Bien que son métier évolue vers la paramétrisation de progiciels ou la composition de composants, il lui reste toujours une partie spécifique à concevoir et à écrire. Pour cette partie là il paraît raisonnable et utile (pour la biodiversité ambiante) de pouvoir utiliser des langages de la famille de ML. L'interopérabilité avec d'autres langages et avec des environnements est alors un point crucial.

```
# #quit ; ;
```
À bientôt...

[20] `nice.sourceforge.net`

Bibliographie

[1] Abadi, M., Cardelli, L., Pierce, B. C., et Rémy, D. Dynamic typing in polymorphic languages. *Journal of Functional Programming 5*, 1 (Janvier. 1995), 111–130.

[2] Abelson, H., et Sussman, G. *Structure and Interpretation of Computer Program.* MIT Press, Boston, 1986.

[3] Aditya, S., et Caro, A. Compiler-directed type reconstruction for polymorphic languages. In *Functional Programming and Computer Architecture* (1993).

[4] Aditya, S., Flood, C. H., et Hicks, J. E. Garbage collection for strongly-typed languages using run-time type reconstruction. In *LISP and Functional Programming* (1994), pp. 12–23.

[5] Aho, A., et Ullman, J. *Fundations of Computer Science.* Freeman and company, 1992.

[6] Ailleret, S. Typage du bytecode Caml. In *Journées Francophones des Langages Applicatifs* (Janvier. 2002), Inria.

[7] Aponte, M.-V., Chailloux, E., Cousineau, G., et Manoury, P. Advanced Programming Features in Objective Caml. In *6th Brazilian Symposium on Programming Languages* (Juin 2002).

[8] Appel, A. Runtime tags aren't necessary. *Lisp and Symbolic Computation* (1989).

[9] Appel, A., et Jim, T. Continuation-passing style, closure-passing style. In *Symposium on Principles of Programming Languages* (1989).

[10] Appel, A., Queen, D. M., et David, B. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture* (1987), G. K. editor, Ed.

[11] Appel, A. W. *modern compiler implementation in C.* Cambridge University Press, 1998.

[12] Appel, A. W. *modern compiler implementation in ML.* Cambridge University Press, 1998.

[13] Armstrong, J. Erlang, bits of history, words of advices of ericsson products, 1998.

[14] Barnier, N., et Brisset, P. FaCiLe : a Functional Constraint Library. In *CICLOPS'01* (2001).

[15] Bartlett, J. F. Compacting garbage collection with ambiguous roots. Tech. Rep. 88/2, Digital Equipement Corporation (WRL), Février. 1988.

[16] Bartlett, J. F. Mostly-copying garbage collection picks up generations and c++. Tech. rep., Digital Equipement Corporation (WRL), Octobre. 1989.

[17] Bayley, P., et Newey, M. Implementing ML on distributed memory multiprocessors. In *Workshop on Languages, Compilers and Run Time Environments for Distributed Memory Multiprocessors* (1992), ACM.

[18] Benton, N., Kennedy, A., et Russel, G. Compiling Standard ML to Java Bytecodes. In *International Conference on Functional Programming* (Septembre. 1998).

[19] Berthomieu, B., et Sergent, T. L. Programming with behaviors in an ML framework, the syntax and semantics of LCS. In *Euporean Symposium On Programming (ESOP'94)* (1994).

[20] Bézivin, J., Cointe, P., Napoli, A., Queinnec, C., et Danvy, O. 20 ans après : où en sont les objets ? In *Colloque scientifique en l'honneur de Jean-François Perrot* (Octobre. 2003), J.-P. Briot, Ed.

[21] Bloch, L. *Initiation à la programmation avec Scheme.* Editions technip, 2001.

[22] Boehm, H., Weiser, M., et Bartlett, J. F. Garbage collection in an uncooperative environment. *Software - Practice and Experience* (Septembre. 1988).

[23] Bosser, A.-G., et Alberti, F. L'expérience SCOL. In *Journées Francophones des Langages Applicatifs* (Janvier. 2003), Inria.

[24] Boulmé, S., Hardin, T., et Rioboo, R. Modules, objets et calcul formel. In *Journées Francophones des Langages Applicatifs* (Janvier. 1999), Inria.

[25] Bourdoncle, F., et Merz, S. Type-checking higher-order polymorphic multi-methods. In *Symposium on Principles of Programming Languages* (Paris, France, 15–17 1997), pp. 302–315.

[26] Bracha, G., Odersky, M., Stouramine, D., et Wadler, P. Making the future safe from the past : Adding Genericity to the Java Programming Language. In *ACM SIGPLAN Conference on Object-Oriented Programming System, Languages and Applications (OOPSLA'98)* (Octobre. 1998).

[27] Brygoo, A., Durand, T., Manoury, P., Queinnec, C., et Soria, M. Un cédérom pour scheme — chacun son entraîneur, un entraîneur pour tous. In *Colloque Internationnal sur les T.I.C.E. d'ingénieur et de l'industrie* (Novembre. 2002).

[28] Capel, C., Chailloux, E., et Eber, J.-M. Applications du toplevel embarqué d'objective caml. In *Journées Francophones des Langages Applicatifs* (Janvier. 2004), Inria. à paraître.

[29] Cardelli, L. The functional asbtract machine. *Polymorphism* (1983).

[30] Chailloux, E. *Compilation des langages fonctionnels : CeML un traducteur ML vers C.* Thèse d'université, Université Paris VII, Novembre. 1991.

[31] Chailloux, E. A Conservative Garbage Collector with Ambiguous Roots for Static Typechecking Languages. In *International Workshop on Memory Management* (Septembre. 1992), Springer-Verlag, Ed., no. 637 in LNCS, ACM SIGPLAN, pp. 218–229.

[32] Chailloux, E. An Efficient Way of Compiling ML to C. In *Workshop on ML and its Applications* (Juin 1992), ACM SIGPLAN.

[33] Chailloux, E. Dynamic Object Typing in Objective Caml. In *Proceedings International Lisp Conference* (Octobre. 2002).

[34] Chailloux, E., Codognet, C., et Codognet, P. Finite Domain Constraints in the ML Functional Language. In *6th International Conference on Tools with Artificial Intelligence* (1994), ACM.

[35] Chailloux, E., et Cousineau, G. Programming Images in ML. In *Workshop on ML and its Applications* (Juin 1992), ACM SIGPLAN.

[36] Chailloux, E., et Foisy, C. Caml-Flight : implantation et applications. In *Journées Francophones des Langages Applicatifs* (Janvier. 1995), Inria.

[37] Chailloux, E., et Foisy, C. A portable Implementation for Objective caml Flight. *Parallel Processing Letters*, 3 (2003). (Journal version of [38].).

[38] Chailloux, E., et Foisy, C. A portable implementation for objective caml flight. In *Second Workshop on High-Level Parallel Programming and Applications* (Juin 2003).

[39] Chailloux, E., et Henry, G. O'jacaré, une interface objet pour entre objective caml et java. In *Langages et Modèles à Objets* (Mars. 2004). à paraître.

[40] Chailloux, E., Manoury, P., et Pagano, B. Types behind the mirror : a proposal for partial ml type reconstruction at runtime. In *Types in Compilation* (Juin 1997).

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

[42] Chailloux, E., Montelatici, R., et Pagano, B. OCAMiL un compilateur Objective Caml pour .NET. In *Conférence Internationale des chercheurs vietnamiens et Francophones en Informatique (RIVF'04)* (Février. 2004).

[43] Chatterjee, S., Blelloch, G. E., et Zagha, M. Scan Primitives for Vector Computers. In *Proceedings Supercomputing'90* (Novembre. 1990), pp. 666–675.

[44] Clinger, W., et Rees, J. Revised^4 report on the algorithmic language scheme. Tech. rep., ACM Lisp Pointers, 1991.

[45] Conchon, S., et Fessant, F. L. Jocaml : Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)* (Palm Springs, CA, USA, 1999).

[46] Cousineau, G. A Brief History of CAML. `http://www.pps.jussieu.fr/~cousinea/Caml/caml_history.html`, 1996.

[47] Cousineau, G. Caml at ENS. In *Functional Languages in the Introductory Computing Curriculum* (Mai 1996).

[48] Cousineau, G., Curien, P. L., et Mauny, M. The categorical abstract machine. In *Functional Programming Languages and Computer Architecture* (1985), Springer-Verlag, Ed.

[49] Cousineau, G., et Mauny, M. *Approche Fonctionnelle de la Programmation*. EdiScience, 1995.

[50] Cridlig, R. Camlot. In *Workshop on ML and its Applications* (Juin 1992), ACM SIGPLAN.

[51] Danelutto, M., DiCosmo, R., Leroy, X., et Pelagatti, S. Parallel functional programming with skeletons : the OCamlP3L experiment. In *Proceedings ACM workshop on ML and its applications* (1998), Cornell University.

[52] David MacQueen, G. P., et Sethi, R. An ideal model for recursive polymorphic types. In *Symposium on Principles of Programming Languages* (Janvier. 1984), pp. 165–174.

[53] Davis, H., Parquier, P., et Séniak, N. Sweet harmony : The talk/c++ connection. In *Lisp and Functional Programming* (1994).

[54] de Rauglaudre, D. camlp4 : Reference manual. Tech. rep., Inria, 2002. `http://caml.inria.fr`.

[55] de Rauglaudre, D., et Mauny, M. Chamau : an ML dialect with quotations, grammars, and extensible syntax. In *Workshop of Compiler Techniques for Application Domain Languages and Extensible Language Models* (Avril. 1996).

[56] Deckner, A., Deckner, N., et Dallot, L. plugin Eclipse pour Objective Caml. Tech. rep., université Paris 6, 2003. Rapport de TER.

[57] Engel, E. *Extensions sûres et praticables du système de tyoes de ML en présence d'un langage de modules et de traits impératifs*. Thèse d'université, université Paris Sud, Mai 1998.

[58] Felleisen, M., et Friedman, D. P. *The Little MLer*. MIT Press, Décembre. 1997.

[59] Findler, R. B., Clements, J., Cormac Flanagan, M. F., Krishnamurthi, S., Steckler, P., et Felleisen, M. Drscheme : A progamming environment for scheme. *Journal of Functional Programming 12*, 2 (March 2002), 159–182.

[60] Finne, S., Leijen, D., Meijer, E., et Jones, S. P. H/Direct : A Binary Foreign Language Interface for Haskell. In *International Conference on Functional Programming* (1998).

[61] Finne, S., Leijen, D., Meijer, E., et Jones, S. P. Calling hell from heaven and heavean from hell. In *International Conference on Functional Programming* (1999).

[62] Foisy, C., et Chailloux, E. Caml Flight : a Portable SPMD Extension of ML for Distributed Memory Multiprocessors. In *Conference on High Performance Functional Computing* (Avril. 1995). available at `ftp ://sisal.llnl.gov/pub/hpfc/papers95/paper16.ps`.

[63] Furuse, J., et Weis, P. Entrée/Sorties de valeurs en Caml. In *Journées Francophones des Langages Applicatifs* (Janvier. 2000), Inria.

[64] Gabriel, R. Objects have failed, Novembre. 2002. `http://www.dreamsongs.com/ObjectsHaveFailedNarrative.html`.

[65] Gabriel, R. P. LISP : Good news, bad news, how to win big. *AI Expert 6*, 6 (1991), 30–39.

[66] Gabriel, R. P. *Patterns of Software : Tales from the software community.* Oxford University Press, 1996. `http://www.dreamsongs.com/Books.html`.

[67] Gamma, E., Helm, R., Johnson, R., et Vlissides, J. *Design Patterns.* Addison Wesley, 1995.

[68] Garrigue, J. Labeled and optional arguments for Objective Caml, Mars. 2001.

[69] Garrigue, J. Code reuse through polymorphic variants. In *Workshop on Foundations os Software Engineering* (nov 2002).

[70] Giacalone, A., Mishra, P., et Prasad, S. Facile : A Symmetric Integration of Concurrent and Functional Programming. *International Journal of Parallel Programming* (1989).

[71] Glaser, H., et Henderson, P. Functional programming and software engineering.

[72] Goldberg, B., et Gloger, M. Polymorphic type reconstruction for garbage collection without tags. In *Conference on Lisp and Functional Programming* (Juin 1992), pp. 53–65.

[73] Gordon, M. J. C., Milner, R., et Wadsworth, C. P. *Edinburgh LCF.* Springer-Verlag, Berlin, 1979.

[74] Gordon, R. *Essential JNI : Java Native Interface.* Prentice Hall, 1999.

[75] Gosling, J. *Specifications du langage Java.* Sun Press, 2000.

[76] Goubault, J. Inférences d'unités physiques en ml. In *Journées Francophones des Langages Applicatifs* (Janvier. 1994), Inria.

[77] Hains, G., et Loulergue, F. Functional Bulk Synchronous Parallel Programming using the BSMLlib Library. In *Constructive Methods for Parallel Programming*, S. Gorlatch, Ed., Advances in Computation : Theory and Practice. Nova Science, august 2002, pp. 165–178.

[78] Hardin, T., et Viguié-Donzeau Gouge, V. *Concepts et outils de programmation.* InterEditions, 1992.

[79] Harper, R., Lee, P., et Pfenning, F. The Fox project : Advanced language technology for extensible systems. Tech. Rep. CMU-CS-98-107, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1998.

[80] Harrison, R. *Abstract Data Types in Standard ML.* John Wiley & Sons, 1992.

[81] Hartel, P. H., Feeley, M., et *et al.* Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. *Journal of Functional Programming 6*, 4 (Jul 1996), 621–655.

[82] Heeren, B., Hage, J., et Swierstra, S. D. Scripting the type inference process. In *International Conference on Functional Programming* (New York, 2003), ACM Press, pp. 3 – 13.

[83] Heeren, B., Leijen, D., et van IJzendoorn, A. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop* (New York, 2003), ACM Press, pp. 62 – 71.

[84] Hirschowitz, T., et Leroy, X. Mixin modules in a call-by-value setting. In *Programming Languages and Systems, ESOP'2002* (2002), D. Le Métayer, Ed., vol. 2305 of *Lecture Notes in Computer Science*, pp. 6–20.

[85] Huet, G. The zen computational linguistics toolkit. Tech. rep., ESSLLI, 2002.

[86] Huet, S. *Le langage Scol.* `www.scol-technologies.org/download.htm`.

[87] Hughes, J. Why Functional Programming Matters. *Computer Journal 32*, 2 (1989), 98–107.

[88] Johnsson, T. Lambda lifting : transforming programs to recursive equations. In *Conference on Functional Programing Languages and Computer Architecture. LNCS 201* (Nancy, 1985), ACM, Springer Verlag.

[89] Jones, R., et Lins, R. *Garbage Collection.* Wiley, 1996.

[90] Kelsey, R., Clinger, W., et (Editors), J. R. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (1998), 26–76.

[91] Kennedy, A. Dimension types. In *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming* (Edinburgh, U.K., 11–13 1994), D. Sannella, Ed., vol. 788, Springer, pp. 348–362.

[92] Kennedy, A., et Syme, D. Design and implementation of generics for the .net common language runtime. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (2001), ACM Press, pp. 1–12.

[93] Krumvieda, C. DML : Packaging High-Level Distributed Abstractions in SML. In *Third International Workshop on Standard ML* (1991).

[94] Le Fessant, F., et Maranget, L. Optimizing pattern-matching. In *International Conference on Functional Programming* (2001), ACM Press.

[95] Leroy, X. Programmation du système Unix en Caml Light. Technical report 147, INRIA, 1992.

[96] Leroy, X. Unboxed objects and polymorphic typing. In *Symposium on Principles of Programming Languages* (1992), ACM Press, pp. 177–188.

[97] Leroy, X. The caml light system, documentation and user's manual, release 0.74. Tech. rep., inria, 1997. on-line version : `http://caml.inria.fr`.

[98] Leroy, X. The effectiveness of type-based unboxing. In *Workshop on Types in Compilation* (Juin 1997).

[99] Leroy, X. *camljava*, 2000. `caml.inria.fr/~xleroy/software.html`.

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

[101] Leroy, X., et Mauny, M. Dynamics in ML. *Journal of Functional Programming 3*, 4 (1993), 431–463.

[102] LinuxMag. Unison : gardez vos documents synchronisés. *Linux Magazine*, 42 (Septembre. 2002).

[103] Loy, M., Eckstein, R., Wood, D., Elliot, J., et Cole, B. *Java Swing.* O'Reilly, 2002.

[104] Maranget, L. Les avertissements du filtrage. In *Journées Francophones des Langages Applicatifs* (2003), Inria.

[105] Mauny, M. Parsers and printers as stream destructors and constructors embedded in functional languages. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture* (1989).

[106] Melhem, W., et Glozic, D. *PDE Does Plug-ins*, 2003.

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

[108] Montelaticci, R., et Chailloux, E. *Documentation and distribution*, 2003. `www.pps.jussieu.fr/~montela/ocamil`.

[109] Morrison, R., Dearle, A., Connor, R. C. H., et Brown, A. L. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems 13*, 3 (July 1991), 342–371.

[110] Netscape. *Netscape Gecko Plug-in API Reference*, 2002. `http://devedge.netscape.com/library/manuals/2002/plugin/1.0/`.

[111] Nikhil, R. *ID Version 90.0, Reference Manual*, 90.1 ed. Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts, USA, 1990.

[112] Norvig, P. A Retrospective on Paradigmes of AI Programming, Janvier. 2003. `www.norvig.com/Lisp-retro.html`.

[113] Nunez, M., Palao, P., et Pena, R. A second year course on data structures based on functional programming. In *Functional Programming Languages in Education* (1995), pp. 65–84.

[114] OBject Technology International, Inc. *Eclipse Platform Technical Overview*, 2003.

[115] Parnas, D. L. Software Engineering Programmes are not Computer Science Programmes. CRL Report 361, Mc Master University, 1998.

[116] Paulson, L. C. *ML for the Working Programmer*, 2nd ed. Cambridge University Press, 1996.

[117] Pecquet, L. Découvrez objectif caml. *Linux Magazine*, 43 (Octobre. 2002).

[118] Pejcic, N., Taguet, A., et Zinovieff, N. Caml applet support plugin for netscape. Tech. rep., université Paris 6, 2002. Rapport de TER.

[119] Peyton Jones, S. *Mise en œuvre des langages fonctionnels de programmation.* Dunod, 1997.

[120] Pottier, F., et Simonet, V. Information flow inference for ml. *ACM Transactions on Programming Languages and Systems 25*, 1 (Janvier. 2003), 117–158.

[121] Puel, L., et Suárez, A. Compiling pattern matching by term decomposition. In *Lisp and Functional Programming* (1990).

[122] Queen, D. M. Modules for standard ml. *Proceedings ACM Synposium on Lisp and Functional Programming* (1984).

[123] Queinnec, C. *Les langages Lisp.* InterEditions, Novembre. 1994.

[124] Queinnec, C., et Chailloux, E. Une expérience de notation de masse. In *TICE 2002* (Novembre. 2002). version complète disponible en `www.infop6.jussieu.fr/licence/2001/cct/cfsreport.ps.gz`.

[125] Queinnec, C., et Weis, P. Programmation applicative, état des lieux et perspectives. *Technique et science informatiques 15*, 7 (1996), 1009–1013.

[126] Quercia, M. *Algorithmique. Cours complet, exercices et problèmes résolus, travaux pratiques.* Vuibert, 2002.

[127] Rémy, D. Using, Understanding, and Unraveling the OCaml Language. In *Applied Semantics. Advanced Lectures. LNCS 2395.*, G. Barthe, Ed. Springer Verlag, 2002, pp. 413–537.

[128] Rémy, D., et Vouillon, J. Objective ML : An effective object-oriented extension to ML. *Theory And Practice of Object Systems 4*, 1 (1998), 27–50. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[129] Reppy, J. CML : A Higher-order Concurrent Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91), Toronto, Canada* (1991), vol. 26 of *SIGPLAN Notices*, ACM Press, pp. 293–305.

[130] Rouaix, F. A Web navigator with applets in Caml. In *Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking* (May 1996), vol. 28, Elsevier, pp. 1365–1371.

[131] Russell, D. *FAD : A Functional Analysis and Design Methodology.* Phd thesis, Computing Laboratory, University of Kent at Canterbury, January 2001.

[132] Salmon, R., et Slater, M. *Computer Graphics : System and concepts.* Addison-Wesley, 1987.

[133] Saura, T. *Etude des modèles d'exécution des langages fonctionnels et impératifs : Application à un évaluateur Scheme.* Thèse d'université, Université Paris 6, Janvier. 1999.

[134] Schiex, T. *Help Est un Lisp Paresseux.* Thèse d'université, université de Toulouse.

[135] Serrano, M. *Vers une programmation fonctionnelle praticable.* Mémoire d'habilitation à diriger la recherche, université de Nice, 2000.

[136] Serrano, M., et Weis, P. $1+1=1$: an optimizing Caml compiler. In *ACM SIGPLAN Workshop on ML and its Applications* (Orlando (Florida, USA), Juin 1994), ACM SIGPLAN, INRIA RR 2265, pp. 101–111.

[137] Serrano, M., et Weis, P. Bigloo : a portable and optimizing compiler for strict functional languages. In *2nd Static Analysis Symposium* (Glasgow, Scotland, Septembre. 1995), Lecture Notes on Computer Science, pp. 366–381.

[138] Siegel, J. *CORBA Fundamentals and Programming.* John Wiley & Sons, 1996.

[139] Signoles, J. Calcul statique des applications de modules paramétrés. In *Journées Francophones des Langages Applicatifs* (2003), Inria.

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

[141] Somogyi, Z., Henderson, F., et Conway, T. Mercury : an efficient purely declarative logic programming language. In *Australian Computer Science Conference* (1995).

[142] Sony Corporation. Communauty place, 1997. `vs.spiw.com/vs`.

[143] Soria, M., Morcrette, M., Brygoo, A., et Paliès, O. *Initiation à la programmation par Word et Excel : Principes et macros.* International Thomson Publishing, 1998.

[144] SPEC. Standard performance evaluation corporation, 1996–2003. `www.spec.org/`.

[145] Steele, G. Objects have not failed, Novembre. 2002. `http://www.dreamsongs.com/ObjectsHaveNotFailedNarr.html`.

[146] Syme, D. ILX : Extending the .NET common IL for functional language interoperability. *Electronic Notes in Theoretical Computer Science 59*, 1 (2001).

[147] Tarditi, D., Lee, P., et Acharya, A. No assembly required : Compiling standard ML to C. *ACM Letters on Programming Languages and Systems 1*, 2 (June 1992), 161–177.

[148] The Joint Task Force on Computing Curricula. Computing Curricula 2001 Computer Science, Final Report. Tech. rep., IEEE Computer Society and Associated for Computing Machinery, Décembre. 2001.

[149] Thompson, S. Higher-order + Polymorphic = Reusable. `www.cs.ukc.ac.uk/pubs/1997/224`, Mai 1997.

[150] Tolmach, A., et Appel, A. W. A Debugger for Standard ML. *Journal of Functional Programming*, 2 (1995).

[151] Tolmach, A. P. Tag-free garbage collection using explicit type parameters. In *Conference on Lisp and Functional Programming* (Juin 1994), pp. 1–12.

[152] Ullman, J. D. *Elements of ML Programming.* Prentice Hall, 1993.

[153] United States Departement of Defense. *The Programming Language Ada Reference Manual.* Springer-Verlag, 1980.

[154] Wadler, P. Why No One Uses Functional Languages. *SIGPLAN Notices 8* (Août. 1998), 23.

[155] WADLER, P. Pizza. In *Principles on Programmationg Languages* (2000).

[156] WEIS, P., APONTE, M. V., LAVILLE, A., MAUNY, M., ET SUAREZ, A. The CAML reference manual. Tech. Rep. 121, INRIA, Septembre. 1990.

[157] WEIS, P., ET LEROY, X. *Le Langage Caml.* InterEditions, 1993. ISBN 2-7296-0493-6.

[158] WILSON, P. R. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management* (Septembre. 1992), Springer-Verlag, Ed., pp. 1–42.

[159] WRIGHT, A. K. Polymorphism for imperative languages without imperative types. Tech. Rep. TR93-200, Rice University Dept. of Computer Science, feb 1993.

[160] X/OPEN GROUP. *DCE 1.1 : Remote Procedure Call*, 1994. `http://www.opengroup.org`.

[161] ZUKOWSKI, J. *Java AWT Reference.* O'Reilly, 1997. on-line version : `www.oreilly.com/catalog/javawt/book/index.html`.

Table des figures

Table des matières