Titres et travaux

Tom Hirschowitz

8 septembre 2009

Table des matières

Structure du document En guise de préambule, cette partie présente une brève description de mon projet de recherche, sur un mode imagé plutôt que technique. Je donne ensuite un bref *curriculum vitae* en partie 2, puis des précisions sur mes résultats en partie 3. Les références complètes de mes publications apparaissent en dernière partie.

1 Programme

Programmation Je situe ma recherche en *programmation*, un domaine qui s'étend traditionnellement de la conception de langages de programmation à l'élaboration de techniques de programmation. La conception de langages de programmation généralistes repose sur le choix d'un ensemble raisonnable d'opérations primitives dont on dote le langage, le but étant de rendre l'écriture de programmes variés la plus efficace possible. Ce choix s'accompagne d'une part de techniques d'*analyse statique* et d'un *schéma de compilation*. Les premières ont pour rôle de détecter certaines erreurs le plus tôt possible dans la vie du programme ; un exemple paradigmatique en est le typage. Le second, le schéma de compilation traduit les programmes en langage machine, avec la charge d'assurer leur efficacité à l'exécution. Cela implique par exemple la définition de transformations de programmes, dont il faut vérifier qu'elles préservent le comportement. On peut ainsi considérer les recherches en équivalences comportementales (bisimulations notamment) comme partie intégrante de la programmation.

La recherche en programmation a fait l'objet de progrès théoriques remarquables par leur utilité pratique. Certains langages poussent ainsi l'analyse statique au point de démontrer des propriétés mathématiques très fines de programmes, par exemple pour borner les erreurs d'arrondi [35] ou garantir l'absence d'erreurs irrattrapables. Au-delà de l'analyse statique, certains outils relient le langage de programmation à un outil d'aide à la démonstration, permettant la démonstration de propriétés trop difficiles pour les outils entièrement automatiques. En compilation, des portions significatives de compilateurs ont été certifiées « par construction » [32].

Ces résultats reposent ultimement sur la même approche mathématique des langages, qui modélise les programmes par des termes dans une grammaire donnée (éventuellement légèrement enrichie, par exemple modulo renommage de variables liées) et l'exécution par une *sémantique opérationnelle*, c'est-à-dire une relation binaire représentant l'évaluation ou l'exécution : $M \to N$ signifie que le programme M retourne la valeur N, ou fait un pas d'exécution pour atteindre l'état N, etc.

Modularité La première phase de mon activité de recherche entre en plein dans cette approche, que je qualifierai d'*opérationnelle*. Le sujet en est la *modularité* dans les langages de programmation, c'est-à-dire les possibilités offertes par un langage donné de construire des programmes par assemblage d'unités plus petites et compréhensibles individuellement, les *modules*. Mon travail a porté sur la notion de *module mixin*, détaillée en partie 3.1 et m'a amené à étudier la construction de définitions récursives dans les langages typés en appel par valeur, au travers de la compilation des modules mixins en définitions récursives, puis des définitions récursives en instructions plus élémentaires.

Mon résultat le plus significatif concerne sans doute la preuve de correction de ce dernier schéma de compilation, parue d'abord comme article de conférence et aujourd'hui en cours de publication [1]. Cette démonstration pourtant longue et difficile ne porte que sur un noyau de langage fonctionnel et rien, dans tout l'outillage opérationnel standard, ne permet d'espérer l'étendre facilement à un langage en taille réelle. En particulier, la seule solution pour cela semble consister à reprendre la démonstration dans son ensemble avec le nouveau langage.

Modularité dynamique Vers 2003, lors de mon arrivée à l'ENS Lyon comme ATER, j'ai commencé à m'intéresser à la *programmation par composants* et sa modélisation à l'aide de calculs de processus [7,23]. Notre travail [4] a porté sur la définition d'un calcul de processus d'ordre supérieur doté de *modularité dynamique*, dans le sens où le langage permet de manipuler les modules au cours de l'exécution, selon des facteurs éventuellement extérieurs, tels que l'interaction avec l'utilisateur. Par exemple, on peut imaginer un navigateur internet remplaçant un module d'affichage multimédia par une version plus récente.

---

[35] Guillaume Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. Thèse de doctorat, Ecole Normale Supérieure de Lyon, Lyon, France, 2006.

[32] X. Leroy. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[7] P. Bidinger and J.-B. Stefani. The kell calculus : operational semantics and type system. In *FMOODS*. Springer-Verlag, 2003.
`http://sardes.inrialpes.fr/papers/files/03-Bidinger-FMOODS.pdf`.

[23] T. Hildebrandt, J. C. Godskesen, and M. Bundgaard. Bisimulation Congruences for Homer — a Calculus of Higher Order Mobile Embedded Resources. Technical Report TR-2004-52, University of Copenhagen, 2004.

Plus de détails sont donnés en partie 3.2 : l'important ici est que nous avons aussi conçu une machine abstraite pour compiler ce langage efficacement. Or, le langage est doté d'une notion de passivation distribuée, dont la compilation a impliqué l'élaboration d'un protocole de communication complexe. De plus, cette compilation ne semble établir qu'une *bisimulation couplée* [37] entre langages source et cible, ce qui est plus faible que la relation attendue en général. Nos tentatives ont révélé qu'une preuve de ce résultat demande un travail d'une bien plus grande ampleur que la preuve de compilation des définitions récursives évoquée plus haut [1], alors que le langage de départ est, en comparaison encore plus réduit.

Sémantique modulaire et passage local-global Ces deux expériences m'ont décidé à chercher une alternative à l'approche opérationnelle. D'une part parce qu'il m'a semblé indésirable de réduire nos résultats à un langage donné, imposant notamment de tout recommencer lors du passage en taille réelle. D'autre part parce que nos démonstrations sont émaillées d'arguments globaux qui encombrent le raisonnement. En particulier, les raisonnements impliquant la notion de *contexte d'évaluation* me semblent nécessiter une compréhension plus profonde.

Depuis 2007, je travaille sur une nouvelle approche des langages de programmation, qu'on appellera ici *sémantique modulaire*, visant à rectifier ces problèmes. L'angle d'attaque adopté consiste à mieux analyser le passage du local au global en programmation. Cette question, fondamentale en topologie, en géométrie et en physique, peut s'interpréter en programmation en définissant chaque construction d'un langage donné par son interaction avec son environnement. Un programme est ainsi vu comme un assemblage, ou un recollement, d'entités élémentaires. Et c'est le comportement de ces entités élémentaires qui détermine celui (ou ceux) du programme, par recollement.

Sémantique modulaire vs. modularité Il faut ici souligner la nuance avec la modularité telle qu'évoquée plus haut. La première différence est qu'en modularité, on cherche à assembler des fragments de programmes à l'aide du langage ; en sémantique modulaire, on le fait de manière éventuellement externe au langage. Par exemple, la notion de contexte d'évaluation n'a généralement qu'une existence externe, elle est rarement utilisée dans le langage. Or, mes premiers travaux avec Garner et Pardon [3] suggèrent l'intérêt d'une notion très proche pour spécifier de manière modulaire les langages de programmation.

Une seconde différence concerne la granularité de la décomposition. La décomposition modulaire au sens traditionnel a une granularité moins fine que la décomposition en entités élémentaires évoquée ci-dessus. Par exemple, un module peut typiquement implémenter une structure mathématique telle que les nombres complexes, les anneaux, les corps, etc. Une entité élémentaire correspond aux constructions mêmes du langage, telles que l'application ou la construction de fonction. Pour pousser un peu ce dernier exemple, la construction de fonction $\lambda x.M$, qui représente la fonction attendant x en argument et exécutant ensuite M, devrait même se décomposer en l'interaction du « contexte » $\lambda x.\Box$ et du terme M [3].

Travaux connexes Mes objectifs rejoignent ceux d'autres chercheurs poussés par des motivations différentes. L'école Montanari a par exemple proposé la notion de *double catégorie* comme point de départ à son étude des langages de programmation concurrents.

---

[37] Joachim Parrow and Peter Sjödin. Multiway synchrinizaton verified with coupled simulation. In *CONCUR '92*, London, UK, 1992. Springer.

Montanari et ses collègues ont aussi révélé la propriété de *décomposition*, dont l'intérêt est que dans une double catégorie la possédant, la bisimulation forte est une congruence — un résultat difficile à obtenir dans certains calculs concurrents. Ils ont ainsi traité de manière satisfaisante le cas du calcul de processus CCS [36]. Néanmoins, les techniques encore proches de l'opérationnel développées dans leurs travaux n'ont pas permis d'obtenir des résultats aussi satisfaisants pour le cas plus complexe du π-calcul, qui utilise notamment la liaison de variable [11]. Melliès [34] a ensuite proposé un modèle plus géométrique utilisant sur les réseaux de preuves de la logique linéaire, qui lui a permis de traiter de manière satisfaisante la logique linéaire multiplicative, un fragment se réduisant par la correspondance de Curry-Howard [27] à un langage doté de liaison de variable, mais ne permettant pas la duplication ou l'effacement de données. C'est Melliès qui a proposé le terme de double catégorie *modulaire* pour ce que Montanari appelle double catégorie satisfaisant la propriété de décomposition.

Méthodes envisagées Mon postulat de départ pour contribuer à ces travaux est qu'il n'y aura pas de modèle modulaire utilisable sans couplage d'une compréhension *géométrique* de la programmation avec une compréhension *algébrique*. Ici, *algébrique* s'entend au sens des théories de Lawvere [30], ou plus généralement des esquisses d'Ehresmann [18]. *Géométrique* signifie informellement que les structures algébriques étudiées doivent présenter une intuition géométrique, ce qui peut s'interpréter formellement comme le fait qu'elles aient des modèles géométriques. Plus ces modèles sont proches du modèle libre, plus le couplage est intime et réussi.

Techniquement, ma proposition actuelle dans cet objectif est transgressive : elle consiste à renier le choix algébrique standard de la *substitution* comme opération primitive de recollement, en l'accusant de voiler la nature géométrique du calcul. La notion de recollement proposée pour remplacer la substitution est celle de substitution *linéaire*. Du point de vue statique déjà, sans parler d'exécution de programme, cette notion favorise l'intuition géométrique : j'ai montré avec Garner et Pardon au cours de la rédaction de notre article [3] que la catégorie des λ-termes modulo renommage des variables liées, avec pour composition la substitution linéaire, se plonge dans une catégorie de réseaux de preuves, objets essentiellement géométriques.

Mes travaux en cours sur une extension de ce plongement à l'exécution des programmes proposent une interprétation géométrique et modulaire du λ-calcul, dont les propriétés restent largement inexplorées, mais qui constitue un début encourageant à mon sens. En plus, les travaux récents de Leinster [31] et Weber [41] sur la notion de *nerf* permettent

[36] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100(1), 1992.

[11] Roberto Bruni and Ugo Montanari. Cartesian closed double categories, their lambda-notation, and the pi-calculus. In *LICS '99*. IEEE Computer Society, 1999.

[34] Paul-André Melliès. Double categories : a modular model of multiplicative linear logic. *Mathematical Structures in Computer Science*, 12, 2002.

[27] W. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, page 479–490. Academic Press Limited, 1980.

[30] F. W. Lawvere. *Functorial semantics of algebraic theories*. Thèse de doctorat, Columbia University, 1963.

[18] Charles Ehresmann. Esquisses et types de structures algébriques. *Bul. Inst. Polit. Iaşi*, XIV(1-2), 1968.

[31] Tom Leinster. Nerves of algebras. Talk at Category Theory '04, 2004.

[41] Mark Weber. Familial 2-functors and parametric right adjoints. *Theory and Applications of Categories*, 18(22) :665–732, 2007.

d'inférer du modèle géométrique une notion algébrique, techniquement une *esquisse* au sens d'Ehresmann [18]. Cette dernière reste pour l'instant difficilement compréhensible et le modèle proposé penche donc encore trop vers le géométrique. Un but important est de renforcer la correspondance entre les deux, par exemple pour permettre l'automatisation du raisonnement. Cela se fera naturellement par la recherche de résultats de complétude démontrant, en gros, que le nouveau modèle n'ajoute pas trop de réductions au modèle opérationnel.

Perspectives Les résultats préliminaires à la réalisation du programme sont la définition d'un cadre algébrique et l'établissement de liens forts avec le modèle opérationnel. A plus long terme, les résultats sont difficiles à prévoir, mais on peut espérer que les techniques évoquées s'appliquent de la définition de langages de programmation à leur compilation en langage machine, en passant par leur typage et l'inférence de types. Quel que soit le champ d'application où elles déboucheront en premier, les premiers résultats prendront sans doute la forme d'une nouvelle démonstration d'un résultat existant, mais résultant d'une méthodologie systématique susceptible de passer à l'échelle.

2 Curriculum vitae

Formation

2003 — Doctorat de l'Université Paris 7, spécialité informatique. Thèse intitulée *Modules mixins, modules et récursion étendue en appel par valeur* et obtenue avec la mention "très honorable" le 18 décembre 2003 devant le jury suivant :

Président	R. Di Cosmo, professeur à l'Université Paris 7,
Directeur de thèse	X. Leroy, directeur de recherche à l'INRIA,
Rapporteurs	G. Boudol, directeur de recherche à l'INRIA, P. Sewell, research fellow, University of Cambridge,
Examinateurs	G. Kahn, directeur scientifique de l'INRIA, P. Lescanne, professeur à l'ENS de Lyon.

2000 — DEA Programmation : sémantique, preuves et langages de l'Université Paris 7, alors dirigé par G. Cousineau, obtenu avec la mention très bien. Stage sous la direction de X. Leroy : typage et compilation des modules mixins pour le langage ML.

1996–2000 — Diplômé de l'Ecole Nationale des Ponts et Chaussées, collège Ingénierie Mathématique et Informatique. Dernière année couplée avec le DEA, le stage de DEA tenant lieu de projet de fin d'études.

Fonctions occupées

2007 –... — CR2 au LAMA (Université de Savoie), dans l'équipe LIMD, dirigée par K. Nour puis par moi-même depuis 2009.

[18] Charles Ehresmann. Esquisses et types de structures algébriques. *Bul. Inst. Polit. Iaşi*, XIV(1-2), 1968.

2004 – 2007	CR2 au LIP (ENS Lyon), dans l'équipe PLUME, dirigée par P. Lescanne puis par D. Hirschkoff.
2003 – 2004	ATER au LIP (ENS Lyon), dans l'équipe PLUME, dirigée par P. Lescanne.
2000 – 2003	Allocataire de recherche au sein du projet CRISTAL à l'INRIA Rocquencourt. Moniteur à l'Université Paris 7 en 2000 – 2001.
2000	Stage de DEA de trois mois au sein du projet CRISTAL, à l'INRIA Rocquencourt, sous la direction de X. Leroy, sur le typage et l'implantation des modules mixins en appel par valeur.
1998 – 1999	Stage d'un an au service R & D du Crédit Commercial de France : conception et implantation en C++ de modèles stochastiques pour les marchés financiers.
1997	Stage de trois mois au Centre de Géologie de l'Ingénieur des Mines de Paris : traitement d'images et programmation en C.

Etudiants

2009	Stage d'observation d'un élève de seconde.
2008-...	Stage de M2 puis thèse de F. Hatat, en codirection avec L. Vuillon (LAMA, UdS)
2007-...	Stage de M2 puis thèse d'A. Pardon, en codirection avec D. Hirschkoff (LIP, ENS Lyon).
2005	Stage de L3 de R. Bardou sur le typage et l'implémentation des modules récursifs en OCaml.
2004	Stage de L3 de S. Lenglet sur le typage de la récursion généralisée.

Enseignement

2008-2009	Cours d'école doctorale à Grenoble, partagé avec P. Hyvernat (LAMA) et D. Duval (IMAG) : *Langages et concepts catégoriques pour la logique et l'informatique.*
2007-2008	Cours au M2IF de l'ENS Lyon, partagé avec Philippe Audebaud : *Théorie de la démonstration.*
2006-2007	Cours au M2IF de l'ENS Lyon : *Catégories en informatique et en logique*, mis à profit pour une autoformation en théorie des catégories.
2005-2006	Cours au M2 d'informatique fondamentale (M2IF) de l'ENS Lyon : *Preuves et types*, mis à profit pour une autoformation en théorie de la démonstration et en particulier à la logique linéaire.

Echanges avec l'étranger

2009	Séjour d'une semaine prévu à l'Institut Max Planck à Bonn, invité par M. Weber, faisant suite au séjour de ce dernier au LAMA en septembre 2009.
2007	Accueil au LAMA de R. Garner, post-doctorant à l'Université d'Uppsala. Son séjour a débouché sur deux articles [3, 12].

2004	Accueil au LIP de S. Fagorzi, étudiante de l'Université de Gênes sous la direction de D. Ancona et E. Zucca.
2001 et 2003	Séjours d'une semaine à l'Université de Gênes, invité par D. Ancona et E. Zucca.
2001 et 2002	Séjours de dix jours à l'Université Heriot-Watt d'Edimbourg, invité par J. B. Wells. Ces séjours ont débouché sur deux articles de conférence [10, 9], avec une version longue du premier [1].

Animation, administration, tâches collectives, projets

2010	Responsable de l'organisation de l'Ecole Jeunes Chercheurs en Informatique Mathématique du GdR du même nom.
2008-2010	Membre du projet ANR Choco.
2006-2008	Responsable du projet ANR ARA-SSIA MoDyFiable.
2007-2008	Membre extérieur de la commission de spécialistes de l'Université Joseph Fourier à Grenoble.
2008-...	Instigateur avec A. Burroni et F. Métayer et administrateur de cats.info, un forum de discussion sur la théorie des catégories.
2007-...	Organisation côté LIMD du séminaire commun Plume-LIMD (LAMA, UdS).
2003-2007	Organisation côté Plume du séminaire commun Plume-Equipe Logique du LAMA (UdS).
2004-2006	Organisation du séminaire du LIP (ENS Lyon).

Participation à des comités de programmes, relecture d'articles, conseils de laboratoire, rédactions de rapports quadriennaux, etc.

3 Recherche

Mon programme étant décrit en partie 1, je m'attache ici à détailler les résultats obtenus dans les différentes phases de ma recherche.

3.1 Modularité

La première phase concerne la modularité dans les langages de programmation. Tous les travaux décrits ici sont en collaboration avec X. Leroy et/ou J. B. Wells (Université Heriot-Watt, Edimbourg).

3.1.1 Contexte

J'ai effectué ma thèse au sein du projet CRISTAL à l'INRIA Rocquencourt, sous la direction de X. Leroy, principal concepteur et implémenteur d'OCaml [33], un langage fonctionnel fortement typé en appel par valeur. Mon travail a porté sur la *modularité* dans les langages fonctionnels en appel par valeur, c'est-à-dire les possibilités offertes par un langage donné de construire des programmes par assemblage d'unités plus petites

[33] Xavier Leroy, Damien Doligez, Jacques Garrigue, and Jérôme Vouillon. The Objective Caml system. Logiciel et documentation disponibles sur le Web, `http://caml.inria.fr/`, 1996-2003.

et compréhensibles individuellement, les *modules*. A l'instar d'un bon nombre d'autres langages, OCaml est doté de constructions dédiées à la modularité, qu'on appelle son *système de modules*. Celui d'OCaml figure parmi les plus évolués mais souffre néanmoins d'au moins deux faiblesses importantes :

Récursion mutuelle Les définitions mutuellement récursives ne peuvent pas s'étendre sur plusieurs modules, ce qui gêne la modularisation dans certains cas [17,13].

Modifiabilité Les mécanismes fournis par le langage pour définir un module incrémentalement à partir d'un autre s'avèrent trop rudimentaires, forçant le programmeur à copier du code manuellement.

Une première observation est que la récursion mutuelle et la modifiabilité sont naturellement présentes dans les langages orientés objets. En effet, grâce à la récursion ouverte, des méthodes mutuellement récursives peuvent être définies dans des classes différentes, qui sont ensuite assemblées par héritage. De plus, l'héritage permet la redéfinition de méthode – la *liaison tardive*, ce qui fournit la modifiabilité.

Pourquoi alors ne pas simplement adopter l'approche orientée objets pour les systèmes de modules ? Essentiellement, parce qu'une classe ne peut contenir que des fonctions (les méthodes). Les variables d'instance et les initialiseurs[1] permettent de contourner le problème, mais sont sources d'erreurs. Il devrait être possible d'alterner naturellement les définitions de fonctions et les définitions calculatoires. Néanmoins, une telle généralisation des classes pose les deux problèmes suivants.

Récursion mutuelle En présence d'héritage, si on permet des définitions arbitraires dans les classes, alors des définitions mutuellement dépendantes complètement arbitraires peuvent apparaître à l'exécution. Dans la plupart des langages en appel par valeur, les définitions récursives sont restreintes statiquement, dans un souci d'efficacité [3] et pour éviter certaines définitions mal fondées. Evidemment, notre système ne doit en aucun cas obliger les concepteurs de langages à abandonner ces propriétés. Il doit donc contrôler les définitions récursives, tant du point de vue du typage que de celui de la sémantique opérationnelle.

Ordre d'évaluation Dans notre système, les classes vont contenir des définitions arbitraires non évaluées, dont l'évaluation sera déclenchée par l'instanciation. Parce que ces définitions sont arbitraires, l'ordre d'évaluation est important. Par exemple, dans une classe c définissant $x = 0$ et $y = x + 1$, x doit être évalué avant y. Ainsi, de façon générale, notre système doit définir un ordre d'évaluation.

Pour résoudre ces problèmes, la notion de *module mixin* semble prometteuse : elle combine la généralité des modules avec certaines constructions directement inspirées des langages orientés objets [12]. Intuitivement, un module mixin est un fragment de code arbitraire, non évalué, qui peut contenir des trous - ses *imports* - et définir des valeurs -

[1] Les initialiseurs sont des méthodes qui sont automatiquement appelées une fois à chaque instanciation de la classe.

[17] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *International Conference on Functional Programming*, pages 262–273. ACM Press, 1996.

[13] K. Crary, R. Harper, and S. Puri. What is a recursive module ? In *PLDI*, pages 50–63. ACM Press, 1999.
`http://www.cs.cmu.edu/~rwh/papers/recmod/recmod.ps`.

[3] Andrew W. Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[12] W. R. Cook. *A Denotational Semantics of Inheritance*. Thèse de doctorat, Department of Computer Science, Brown University, 1989.

ses *exports*. Les trous sont comblés lors de la composition de deux modules mixins, où les exports de l'un sont reconnus comme les imports de l'autre, et *vice versa*. Lorsque tous les trous sont remplis, le module mixin obtenu peut être *instancié* en un module ordinaire, ce qui déclenche son évaluation. Les exports d'un module mixin sont en liaison tardive et peuvent être redéfinis jusqu'à l'instanciation. Enfin, les exports d'un module mixin peuvent être mutuellement récursifs. Les modules mixins semblent donc combler les défauts évoqués ci-dessus.

La notion de module mixin n'est pas neuve : G. Bracha [9] l'a introduite il y a déjà dix ans, suivi par D. Duggan et C. Sourelis [17] et M. Flatt et M. Felleisen [20]. Cependant, ces travaux restreignent encore l'héritage à certaines sortes de composantes de modules, typiquement les fonctions : même si les modules peuvent contenir d'autres sortes de définitions, seules les définitions de fonctions sont concernées par l'héritage.

D. Ancona et E. Zucca [2,1] et J. B. Wells et R. Vestergaard [42] étendent indépendamment l'héritage à toutes les sortes de définitions, mais leur travail ne s'applique qu'aux langages paresseux tels que Haskell [22], donc pas aux langages en appel par valeur tels que OCaml. En particulier, les langages paresseux acceptent toutes les définitions récursives sans restriction et l'ordre d'évaluation est géré de manière très différente des langages par valeur : c'est l'évaluation du résultat final qui le détermine (cela revient à prendre le calcul par la fin, plutôt qu'à suivre une suite d'instructions).

Enfin, globalement, aucun de ces travaux ne considère le problème de la compilation des modules mixins.

Le cœur de ma contribution à l'étude des modules mixins consiste en une étude poussée des définitions récursives, réalisée sur un λ-calcul avec définitions récursives, $\lambda_\circ$ [15, 10, 1]. Pour résumer, j'ai d'abord défini un système de types pour $\lambda_\circ$, qui garantit l'absence d'erreurs à l'exécution, puis formalisé l'équivalent du schéma de compilation d'OCaml comme une traduction de $\lambda_\circ$ vers un langage fonctionnel avec tas, dont j'ai démontré la correction [1]. Ensuite, j'ai proposé trois langages typés de modules mixins, différents surtout par leur gestion de l'ordre d'évaluation. J'ai aussi démontré que le premier de ces trois langages se compile vers des programmes bien typés en $\lambda_\circ$, ce qui assure l'absence de définitions récursives indésirables.

La fin de cette partie est consacrée au survol de ces travaux, du haut vers le bas : je commence par résumer les deux approches proposées pour gérer l'ordre d'évaluation, ainsi que le typage dans chaque cas et je poursuis en décrivant le travail sur $\lambda_\circ$.

---

[9] Gilad Bracha. *The Programming Language Jigsaw : Mixins, Modularity and Multiple Inheritance*. Thèse de doctorat, université de l'Utah, 1992.

[17] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *International Conference on Functional Programming*, pages 262–273. ACM Press, 1996.

[20] Matthew Flatt and Matthias Felleisen. Units : cool modules for HOT languages. In *PLDI*, pages 236–248. ACM Press, 1998.

[2] Davide Ancona. *Modular Formal Frameworks for Module Systems*. Thèse de doctorat, université de Pise, 1998.

[1] D. Ancona and E. Zucca. A primitive calculus for module systems. In Gopalan Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *LNCS*, pages 62–79. Springer-Verlag, 1999.

[42] J. B. Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. In *European Symposium on Programming*, volume 1782 of *LNCS*, pages 412–428. Springer-Verlag, 2000.

[22] The Haskell language. `http://www.haskell.org`.

3.1.2 Ordre d'évaluation

Déterminer l'ordre d'évaluation à l'instanciation Pour résoudre le problème de l'ordre d'évaluation, les langages CMS_v [11, 2] et MM [15, 9] calculent un ordre valide au moment de l'instanciation, sur la base des dépendances entre les définitions. CMS_v est un langage noyau de modules mixins, dont la sémantique est définie par traduction vers un λ-calcul avec récursion (voir plus loin). MM propose plus d'opérateurs, permet les définitions anonymes (invisibles depuis l'extérieur). De plus, il est défini par une sémantique opérationnelle, ce qui facilite le raisonnement.

Dans l'exemple de la classe c ci-dessus définissant $x = 0$ et $y = x + 1$, le fait que y dépende de x le place après dans l'ordre calculé lors de l'instanciation de c. Un raffinement de cette proposition consiste à permettre au programmeur de spécifier manuellement l'ordre calculé. Lorsque les dépendances entre définitions sont soit inconnues, soit susceptibles de changer dans des versions futures, il devient possible de maintenir un ordre entre elles. Cette solution est très puissante, puisqu'elle permet le réordonnancement *a posteriori* des définitions.

Du point de vue du typage, une simple adaptation du typage des modules ou des classes s'avère insuffisante, comme le montre l'exemple suivant, dans une syntaxe proche de celle d'OCaml [33] :

```
mixin A = import
  val y : int
export
  val x = y + 1
end

mixin B = import
  val x : int
export
  val y = x * 2
end
```

Le module mixin `A` importe une valeur `y` et exporte la définition `x = y + 1`. Le module mixin `B` est complémentaire : il importe une valeur `x` et définit `y = x * 2` en fonction de celle-ci.

Naïvement, on pourrait s'attendre à ce qu'un typage de la forme

```
mixin A : import val y : int
          export val x : int end
mixin B : import val x : int
          export val y : int end
```

suffise à garantir la sûreté d'exécution du programme. Evidemment, l'exemple choisi montre qu'il n'en est rien : l'information manque dans ces types pour deviner l'apparition de la définition récursive mal formée lors de la composition.

Il est donc difficile de définir un système de types vérifiant statiquement la validité de programmes en présence de modules mixins. La solution proposée [15, 9, 11, 2] consiste à inclure dans les types de modules mixins des informations fines sur les dépendances entre composantes. On distingue en particulier une dépendance forte en une variable `x`,

[33] Xavier Leroy, Damien Doligez, Jacques Garrigue, and Jérôme Vouillon. The Objective Caml system. Logiciel et documentation disponibles sur le Web, `http://caml.inria.fr/`, 1996-2003.

nécessitant le calcul de la valeur de `x`, d'une dépendance faible, qui n'a besoin que de la variable. Par exemple, `x + 1` dépend fortement de `x`, alors que la fonction `fun y -> y + x` dépend faiblement de `x`.

Les définitions récursives mal formées apparaissent sous forme de certains cycles de dépendance dans les types. Dans les dernières versions du système, une notion de sous-typage sur les dépendances rend le typage plus flexible. Une syntaxe concrète appropriée évite au programmeur d'avoir à écrire lui-même les informations de dépendances dans les interfaces de modules mixins. Ce système pose les bases théoriques d'un typeur pour un langage de modules mixins pour ML.

Un langage plus réaliste, incluant des définitions de types à la ML, a été défini [16]. Un typeur prototype basé sur le compilateur OCaml a été réalisé, dans lequel de nombreux exemples significatifs ont été programmés, révélant certaines limites du langage. Notamment, la présence de graphes de dépendances dans les types de modules mixins alourdit leur syntaxe et pose le problème d'une restriction plus praticable de MM. Le programme et les exemples sont disponibles à l'adresse `http://www.lama.univ-savoie.fr/~hirschowitz/software/type_components.tar.gz`.

Déterminer l'ordre d'évaluation à la composition Le langage MM et notamment son système de typage mettent en œuvre des formalismes puissants, mais relativement complexes. En effet, lors de l'instanciation d'un module mixin, si l'approximation sur les dépendances permet de trouver un ordre d'évaluation valide, alors le système utilise cet ordre pour générer le module correspondant. Cette puissance vient au prix de l'utilisation de graphes de dépendances, à la fois au niveau de la sémantique opérationnelle et au niveau du typage. Ceci semble compromettre l'intégration de MM à un langage de programmation réel tel que ML.

Le langage *Mix* [8] simplifie grandement MM, sans trop perdre en pouvoir d'expression. L'idée consiste à choisir un ordre d'évaluation plus tôt que dans MM. Dans *Mix*, l'ordre des définitions dans un module mixin est respecté jusqu'à l'instanciation et la composition de deux modules mixins calcule un ordre compatible avec ceux de ses arguments. Ce calcul, du fait des contraintes plus fortes, est quasi-linéaire en le nombre de définitions, alors qu'il est quadratique en MM. De plus, la gestion de l'ordre d'évaluation est beaucoup plus simple en *Mix*, puisque l'ordre de définition n'est jamais modifié. Enfin, le typage de *Mix* est aussi nettement plus simple que celui de MM : il ne nécessite pas d'information de dépendances.

Mix n'a pas encore été enrichi avec du sous-typage et des définitions de types à la ML : le passage à une implantation réaliste permettant de vérifier que le résultat est viable reste donc à faire. Néanmoins, *Mix* semble un bon candidat pour intégrer un système de modules mixins au langage ML.

3.1.3 Typage et compilation de la récursion

Les parties précédentes présentent mon travail sur la définition et le typage des modules mixins en appel par valeur. Reste à examiner le problème de leur compilation. C'est ce problème qui m'a conduit à m'intéresser aux définitions récursives.

On a vu que la notion de module mixin est une généralisation de la notion de classe. On pourrait donc imaginer que la compilation des classes s'étende à celle des modules mixins, mais cela s'avère difficile, comme le montre l'exemple suivant.

Considérons le module mixin

```
mixin C = import
```

```
export
  val x = 0
  val y = x + 1
end
```

Dans la représentation classique des objets par des enregistrements récursifs, qui permet de préserver la compilation séparée, `C` est représenté par le foncteur suivant :

```
module C = functor Self ->
  struct
    val x = 0
    val y = Self.x + 1
  end
```

Cette fonction prend en argument un module `Self`, qui représente la valeur finale de l'instance. Elle renvoie le module obtenu en fonction de cette valeur finale. Une instance de `C` est un point fixe de cette fonction.

Le problème est qu'ici, l'opérateur standard `fix` de point fixe en appel par valeur n'est pas assez fin pour trouver le point fixe. En effet, `fix C` se réduit en

```
struct
  val x = 0
  val y = (fix C).x + 1
end
```

qui requiert à son tour l'évaluation de `fix C`, générant ainsi une chaîne de réduction infinie.

Point fixe local et récursion étendue Au lieu de représenter les modules mixins par des foncteurs, nous proposons de répartir l'abstraction sur chaque composante. En d'autres termes, un module mixin est représenté par un module dont les composantes sont des fonctions. Par exemple, le module mixin `C` est représenté par le module

```
module C = struct
  val x = fun () -> 0
  val y = fun x () -> x + 1
end
```

La définition de `y` est abstraite par rapport à la valeur finale de `x`, qu'on lui fournira seulement lors de l'instanciation. De plus, chaque définition prend un argument non utilisé, qui sert à suspendre le calcul jusqu'à l'instanciation. Dans cette nouvelle représentation, l'instanciation consiste en un *point fixe local* de ce module. Ainsi, l'instanciation de `C` est compilée en

```
struct
  val x = C.x ()
  val y = C.y x ()
end
```

qui s'évalue comme attendu.

Un problème apparaît pour compiler les définitions de fonctions mutuellement récursives. Par exemple, considérons le module mixin

```
mixin Nat = import
export
  val even x = x = 0 or odd (x-1)
  val odd x = x > 0 and even (x-1)
end
```

Son instanciation donne

```
struct
  val rec even = Nat.even odd ()
      and odd = Nat.odd even ()
end
```

Le problème de ce programme est qu'il est refusé par la plupart des langages en appel par valeur. En effet, comme mentionné précédemment, ces langages restreignent généralement les définitions récursives statiquement [3]. Notamment, les applications de fonctions telles que `Nat.even odd ()` sont refusées comme membres droits de définitions récursives.

Pour compiler les modules mixins, nous avons donc besoin d'un langage doté d'un mécanisme de définitions récursives plus puissantes, capable de gérer les définitions ci-dessus. De tels mécanismes ont déjà été proposés dans d'autres contextes, notamment les définitions récursives monadiques d. L. Erkök et J. Launchbury [19], les langages en appel par valeur de G. Boudol [8] et les travaux de Z. Ariola et al. sur la récursion [5,6,4].

Le point capital d'une telle construction dans notre contexte est l'efficacité du code produit. Il est possible d'encoder les définitions récursives voulues au moyen de références initialisées avec des valeurs par défaut, puis mises à jour. Mais cette méthode introduit une indirection supplémentaire à chaque accès aux valeurs obtenues.

Notre proposition est le langage $\lambda_\circ$ [15, 10, 1], pour lequel la méthode de compilation des définitions récursives en OCaml [33] s'applique presque directement et permet d'éviter toute indirection supplémentaire. Dans ma thèse [15] et dans l'article HOSC [1], je formalise cette méthode et prouve sa correction. Le papier antérieur [10] suit la même démarche, mais pour un langage moins puissant que $\lambda_\circ$ vis-à-vis des définitions récursives.

Typage Dans le cadre de la compilation des modules mixins, il est utile de doter $\lambda_\circ$ d'un système de typage suffisamment expressif, notamment pour s'assurer de la validité de la compilation ou de transformations de programmes. Mais plus largement, le typage fin des définitions récursives pose aussi problème dans le cadre des objets, ou bien des modules récursifs (non mixins).

---

[3] Andrew W. Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[19] Levent Erkök and John Launchbury. Recursive monadic bindings. In *International Conference on Functional Programming*, pages 174–185, 1999.

[8] G. Boudol. The recursive record semantics of objects revisited. In *ESOP*, LNCS. Springer-Verlag, 2001.
`ftp://ftp-sop.inria.fr/mimosa/personnel/gbo/safe-rec-obj.ps.gz`.

[5] Zena M. Ariola and Jan Willem Klop. Cyclic lambda graph rewriting. In *Logic in Computer Science*, pages 416–425, 1994.

[6] Zena M. Ariola and Jan Willem Klop. Lambda calculus with explicit recursion. *Information and Computation*, 139 :154–233, 1997.

[4] Zena M. Ariola and Stefan Blom. Skew confluence and the lambda calculus with letrec. *Annals of pure and applied logic*, 117(1-3) :95–178, 2002.

[33] Xavier Leroy, Damien Doligez, Jacques Garrigue, and Jérôme Vouillon. The Objective Caml system. Logiciel et documentation disponibles sur le Web, `http://caml.inria.fr/`, 1996-2003.

Ainsi, G. Boudol [8] propose un système de typage pour une représentation des objets à base de références. Son système n'est pas assez expressif pour typer les définitions récursives nécessaires à la compilation des modules mixins, mais est à la base de notre proposition.

D. Dreyer [16] a récemment proposé un système de types complexe, impliquant une gestion spéciale de l'accès aux définitions récursives, mais s'avérant très expressif.

L'article ESOP '02 [11], sa version journal [2] et ma thèse [15] proposent deux systèmes étendant celui de G. Boudol et permettant de typer les définitions récursives voulues, sans gestion spéciale des définitions récursives.

3.2 Modularité dynamique

Vers la fin de ma thèse, après avoir étudié la modularité pendant trois ans, j'ai été intrigué par l'idée de *programmation par composants*. En effet, le slogan de la programmation par composants est très proche de la modularité puisqu'il s'agit aussi de décomposer les programmes en entités élémentaires plus petites et compréhensibles individuellement. Sauf qu'ici on ne veut plus seulement construire les programmes par assemblage de telles entités, on veut aussi les déployer, les maintenir, les reconfigurer, etc. En d'autres termes, la structure modulaire doit persister à l'exécution et rester modifiable soit de manière autonome par le programme lui-même, soit par l'intervention extérieure d'un administrateur ou d'un utilisateur.

Avec mes collègues du moment, D. Hirschkoff et D. Pous à l'ENS Lyon et A. Schmitt et J.-B. Stefani à Grenoble, j'ai donc commencé à m'intéresser à ce que nous avons appelé la *modularité dynamique* et qui a donné lieu au projet ANR MoDyFiable (voir en partie 2).

Notre but était d'isoler un langage de programmation aussi restreint que possible, permettant d'utiliser les idiomes des praticiens de la programmation par composants. L'équipe de Schmitt et Stefani à Grenoble apportait son expertise en la matière, puisqu'elle est à l'origine du modèle de composants Fractal [10]. L'adoption d'une approche par langage de programmation présente l'avantage de poser très nettement les questions de la compilation efficace et de l'analyse statique. Notre cahier des charges peut se résumer en quatre points plutôt informels :

- Le langage doit être adapté à la programmation concurrente et distribuée.
- Il doit de plus être doté d'une théorie comportementale bien comprise, c'est-à-dire d'une notion adéquate d'équivalence entre programmes. Un critère raisonnable pour une telle notion est qu'elle se prête à la vérification automatique.
- Le langage doit être dynamiquement modulaire dans le sens ci-dessus.
- Le langage doit être implémentable. Un critère technique est qu'il doit permettre le prototypage rapide d'applications distribuées avec une efficacité raisonnable à l'exécution.

Avant mon arrivée dans ce groupe, Schmitt et Stefani avaient déjà proposé le *kell*

---

[8] G. Boudol. The recursive record semantics of objects revisited. In *ESOP*, LNCS. Springer-Verlag, 2001.
`ftp://ftp-sop.inria.fr/mimosa/personnel/gbo/safe-rec-obj.ps.gz`.

[16] Derek Dreyer. A type system for well-founded recursion. In *31st POPL*. ACM Press, 2004.

[10] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in Java. In *Component-based Software Engineering*, volume 3054 of *LNCS*. Springer-Verlag, 2004.

calcul [39,7], un calcul de processus avec *localités passivables*. Cela signifie d'abord que le langage est concurrent, grâce à la présence d'un opérateur de composition parallèle : on écrit $P \mid Q$ pour le programme exécutant P et Q de manière concurrente. De plus, on peut nommer des fragments de programmes, ou leur assigner une *localité* : ainsi, $a[P]$ désigne l'instruction d'exécuter P *en* a. Une localité est assimilée à un composant. Enfin, on peut passiver de tels composants ; on a en particulier la règle de réduction

$$a[P] \mid (a[X] \triangleright Q) \longrightarrow Q\{P/X\},$$

où la requête $a[X]$ *passive* le composant situé en a. Le calcul est aussi doté de primitives de communication locale par message.

Sa lacune principale en vue d'une modélisation d'un cadre tel que Fractal est l'absence de *partage* de composant, qui gêne la modularité. Pour prendre un exemple simpliste, il est impossible à deux composants différents d'utiliser le même composant, disons, d'impression, ou de log (écriture de messages système). Impossible est un peu fort ; disons plutôt que ça demande des contorsions excessives.

Notre première tentative [7] consiste à distinguer la hiérarchie de *gestion*, ou de *possession* (*ownership* en anglais), des composants du graphe de *communication*. L'imbrication syntaxique représente la hiérarchie de gestion : il faut être gestionnaire d'un composant pour pouvoir le passiver. La communication se fait ensuite par pointeurs : la présence d'un processus $*a$ permet de communiquer avec a. Par exemple, en simplifiant en peu, dans le programme

$$a[b[x()]|c[P \mid d[*a \mid x\langle\rangle]]], \tag{1}$$

a gère b, mais seul d communique avec b. Le programme se réduit par exemple à $a[b[]|c[P \mid d[*a]]]$ après communication sur le canal x.

Le problème de ce langage, qui nous a poussés à proposer une alternative [4], est que les pointeurs $*a$ sont en liaison dynamique. Par exemple, dans le programme (1) ci-dessus, si P se réduit à $a[x()]$, c'est avec cette nouvelle occurrence de a que d pourra communiquer et uniquement avec elle. La liaison dynamique étant généralement considérée comme indésirable, nous avons proposé un autre langage insipré du π-calcul [36] et du Join calcul [21]. C'est en gros le π-calcul avec des localités passivables, où l'opérateur de restriction ν est confiné : il ne traverse pas sa localité parente.

Ce travail ne représente qu'une étape vers la conception d'un langage obéissant au cahier des charges ci-dessus : le côté dynamique de la modularité y est largement négligé. Mais il fait déjà se rencontrer modularité et concurrence, ce qui est un début. Nous l'avons donc étudié, en commençant par le traduire vers une machine abstraite, que D. Pous a implémentée comme une bibliothèque OCaml. Et c'est là que ça se corse : démontrer la correction de cette traduction s'est avéré demander un travail d'une ampleur dépassant de loin l'enjeu. Pour ce petit langage restreint, en un an nous aurions peut-être obtenu une démonstration informelle de correction, sans le moindre espoir de passage à l'échelle.

---

[39] Jean-Bernard Stefani. A calculus of higher-order distributed components. Rapport de recherche RR-4692, Inria, Janvier 2003.

[7] P. Bidinger and J.-B. Stefani. The kell calculus : operational semantics and type system. In *FMOODS*. Springer-Verlag, 2003.
`http://sardes.inrialpes.fr/papers/files/03-Bidinger-FMOODS.pdf`.

[36] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100(1), 1992.

[21] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the Join-calculus. In *POPL '96*, pages 372–385. ACM Press, 1996.

En plus, nous ne pouvions espérer qu'un résultat de correction relativement faible, basé sur la notion de bisimulation *couplée* [37]. Notre article [4] s'arrête sur ce constat.

Vis-à-vis d'un tel problème, il n'est pas déraisonnable pour un informaticien de se satisfaire de la conviction intime que la compilation est correcte en attendant que les outils mathématiques rattrapent la technologie. Mais ce problème m'a intéressé et entraîné en dépit de ma condition d'informaticien, ce qui nous mène vers la troisième phase de mes recherches et mon programme de *sémantique modulaire*.

3.3 Sémantique modulaire

Le programme étant décrit en détail en partie 1, je me contenterai ici de décrire les premiers résultats obtenus, qui concernent l'utilisation des catégories *symétriques monoïdales fermées* (SMC) pour représenter de manière algébrique la syntaxe avec variables liées. Ces travaux sont en collaboration avec R. Garner (Universités d'Uppsala en Suède, puis de Cambridge) et mon étudiant A. Pardon.

Nous reformulons d'abord un résultat de Trimble [40] en utilisant une présentation plus simple due à Hughes [28]. Ce résultat donne une caractérisation de la categégorie SMC initiale engendrée par une *théorie* SMC. Cela donne lieu à une adjonction entre théories et modèles, dans la lignée des travaux de Lawvere sur les théories algébriques [30]. Une théorie SMC est obtenue en ajoutant des équations à une *signature* SMC. Une telle signature consiste en un ensemble de *sortes* X, accompagné d'un graphe, dont les sommets sont des *formules* sur X, comme définies par la syntaxe :

$$A, B, \ldots \in \mathscr{F}(X) \quad ::= \quad x \mid I \mid A \otimes B \mid A \multimap B \qquad\qquad x \in X,$$

où $\otimes$ s'appelle *tenseur* et $\multimap$ *implication (linéaire)*. Ces formules sont celles de la logique linéaire multiplicative intuitionniste (IMLL).

On pense à une signature comme donnant des opérations. Par exemple, une signature naïve pour le λ-calcul comprend une sorte t des *termes* et deux opérations :

$$t \otimes t \xrightarrow{\quad\cdot\quad} t \qquad\qquad t \multimap t \xrightarrow{\quad\lambda\quad} t \tag{2}$$

pour application et abstraction. Sauf qu'en réalité, c'est le λ-calcul linéaire que cette signature engendre. Nous verrons plus bas comment obtenir la variante non linéaire.

La catégorie SMC $S(\Sigma)$ engendrée par une telle signature Σ, dans notre présentation, a pour objets les formules et pour morphismes une variante des *réseaux de preuve* de la logique linéaire. Les morphismes sont en correspondance étroite avec les termes modulo renommage de variable liées, au sens usuel. Il y a cependant deux différences importantes :

- La composition dans $S(\Sigma)$ correspond à la substitution *linéaire*, par opposition à la substitution usuelle.
- Il y a beaucoup plus dans $S(\Sigma)$ que les seuls termes.

---

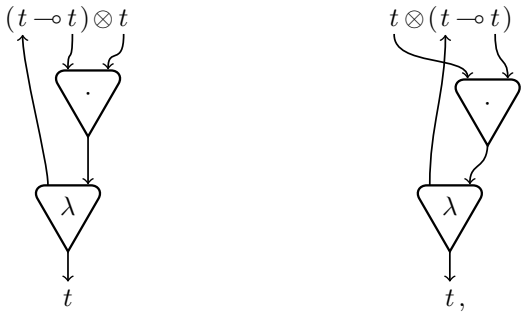
[37] Joachim Parrow and Peter Sjödin. Multiway synchrinizaton verified with coupled simulation. In *CONCUR '92*, London, UK, 1992. Springer.

[40] Todd H. Trimble. *Linear logic, bimodules, and full coherence for autonomous categories.* Thèse de doctorat, Rutgers University, 1994.

[28] Dominic J. D. Hughes. Simple free star-autonomous categories and full coherence. ArXiv Mathematics e-prints, math/0506521, June 2005.

[30] F. W. Lawvere. *Functorial semantics of algebraic theories.* Thèse de doctorat, Columbia University, 1963.

Pour illustrer le second point, $S(\Sigma)$ contient par exemple une sorte de contexte d'évaluation contrôlé. En prenant pour Σ la signature (2) ci-dessus, on obtient par exemple deux versions du contexte $\lambda x.\, \square_1\, \square_2$:



l'une forçant x à être utilisée dans $\square_1$, l'autre dans $\square_2$. L'article [3] fournit de nombreux exemples explorant $S(\Sigma)$, ainsi qu'un résultat de factorisation surprenant, bien que complètement inutile pour l'instant.

La théorie s'affine en fait un peu plus avec l'introduction d'*équations* entre morphismes de $S(\Sigma)$, ce qui mène à la notion de *théorie* (SMC). Cette dernière permet de retrouver le λ-calcul, dans le sens où on définit une théorie Λ, telle que dans la SMC catégorie $S(\Lambda)$ on obtient :

Théorème 1. *Les morphismes $I \longrightarrow t$ sont en bijection avec les λ-termes clos.*

Cette théorie Λ comporte deux sortes, t pour *termes* et v pour *variables*, comme opérations

$$(v \multimap t) \xrightarrow{\lambda} t \qquad (t \otimes t) \xrightarrow{\cdot} t$$

$$v \xrightarrow{d} t \qquad v \xrightarrow{c} v \otimes v \qquad v \xrightarrow{w} I.$$

On ajoute ensuite les axiomes faisant de v un comonoïde commutatif : la *contraction* c est associative et commutative et l'*affaiblissement* w est son neutre. Cette théorie s'inspire de travaux antérieurs [15,25,26], mais les internalise à une catégorie SMC donnée, un peu dans le style des travaux de Montanari et al. [11]. On distingue la sorte v des variables, qu'on peut utiliser plusieurs fois, de la sorte t des termes, qui sont linéaires.

Une information à retirer de ce résultat est que la structure de catégorie SMC prend en charge la *portée* des variables liées, au sens où $S(\Lambda)$ ne contient pas de termes où des variables liées s'échappent de leur portée. Par exemple :

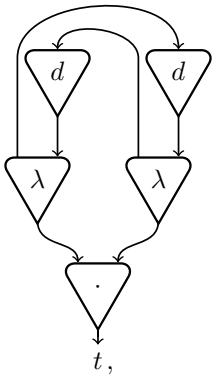
---

[15] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In *TLCA '95*, volume 902 of *LNCS*. Springer, 1995.

[25] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *LICS '99*. IEEE Computer Society, 1999.

[26] Furio Honsell, Marino Miculan, and Ivan Scagnetto. pi-calculus in (co)inductive-type theory. *Theor. Comput. Sci.*, 253(2), 2001.

[11] Roberto Bruni and Ugo Montanari. Cartesian closed double categories, their lambda-notation, and the pi-calculus. In *LICS '99*. IEEE Computer Society, 1999.



qui correspondrait peut-être au terme $(\lambda x.y)(\lambda y.x)$, où x et y seraient liées dans tout le terme, n'est pas un morphisme correct.

Nous appliquons ensuite notre technique aux *bigraphes* de O. H. Jensen et R. Milner [29], en donnant une traduction de leurs *signatures bigraphiques* en des théories SMC. Nous comparons ensuite la catégorie de bigraphes $M(K)$ engendrée par une signature bigraphique K et la catégorie SMC $S(T_K)$ engendrée par la théorie SMC associée T_K. Nous construisons en particulier un foncteur $\mathscr{T}\colon M(K) \to S(T_K)$, fidèle et essentiellement injectif sur les objets. Ce foncteur n'est pas plein, ce qui signifie que, localement, la notion de portée imposée par les bigraphes est plus stricte que celle imposée par la structure SMC. Malgré cela, nous montrons que la notion globale de portée est la même :

Théorème 2. *Pour tout objet U de $M(K)$, on a une bijection $S(T_K)(I, \mathscr{T}(U)) \cong M(K)(I, U)$.*

Autrement dit, $S(T_K)$ a les mêmes programmes que $M(K)$, mais plus de fragments de programmes.

Ce travail montre de manière encourageante qu'il est possible, pour les aspects statiques de la programmation au moins, de coupler une approche algébrique avec une compréhension géométrique.

3.4 Jeux graphiques

En parallèle à mon programme de sémantique modulaire, avec mon père André, mathématicien à Nice, je me suis intéressé pour des raisons évidentes de proximité à la thèse de mon frère Michel [24], aujourd'hui chercheur en informatique au CEA. Cela a mené d'abord à un article de conférence introduisant ses travaux [6], puis à l'idée nouvelle de *jeu graphique*, proposée indépendamment par Delande et Miller [14], que nous avons pour l'instant appliquée à divers fragments de la logique linéaire. Mon étudiant F. Hatat travaille actuellement sur la construction de *catégories de réponses* [38] libres à l'aide de jeux graphiques.

---

[29] Ole H. Jensen and Robin Milner. Bigraphs and mobile processes (revised). Technical Report TR580, University of Cambridge, 2004.

[24] Michel Hirschowitz. *Jeux abstraits et composition catégorique.* Thèse de doctorat, Université Paris 7, 2004.

[14] Olivier Delande and Dale Miller. A neutral approach to proof and refutation in MALL. In *LICS '08*, 2008.

[38] Peter Selinger. Control categories and duality : on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2) :207–260, 2001.

4 Références

Mes articles sont accessibles en ligne à l'adresse `http://hal.archives-ouvertes.fr/aut/Tom+Hirschowitz/`.

Articles dans des revues avec comité de lecture

[1] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. *Higher-Order and Symbolic Computation*, 2009. To appear.

[2] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. *ACM Transactions on Programming Languages and Systems*, 27(5) :857 – 881, 2005. doi :10.1145/1086642.1086644.

Articles dans des actes de conférences avec comité de programme

[3] Richard Garner, Tom Hirschowitz, and Aurélien Pardon. Variable binding, symmetric monoidal closed theories, and bigraphs. In *CONCUR 2009 – Concurrency Theory*, volume 5710 of *Lecture Notes in Computer Science*. 2009. Une introduction aux articles plus techniques [12, 14].

[4] Daniel Hirschkoff, Aurélien Pardon, Tom Hirschowitz, Samuel Hym, and Damien Pous. Encapsulation and Dynamic Modularity in the Pi-Calculus. In *Proceedings of the First Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2008)*, volume 241 of *Electronic Notes in Theoretical Computer Science*, pages 85 – 100. 2009. doi :10.1016/j.entcs.2009.06.005.

[5] André Hirschowitz, Michel Hirschowitz, and Tom Hirschowitz. Contraction-free proofs and finitary games for Linear Logic. In *Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009)*, volume 249 of *Electronic Notes in Theoretical Computer Science*, pages 287–305. 2009. doi :10.1016/j.entcs.2009.07.095.

[6] Michel Hirschowitz, André Hirschowitz, and Tom Hirschowitz. A theory for game theories. In *FSTTCS 2007 : Foundations of Software Technology and Theoretical Computer Science FSTTCS 2007 : Foundations of Software Technology and Theoretical Computer Science*, volume 4855 of *Lecture Notes in Computer Science*, pages 192–203. Springer, 2007. doi :10.1007/978-3-540-77050-3.

[7] Daniel Hirschkoff, Tom Hirschowitz, Damien Pous, Alan Schmitt, and Jean-Bernard Stefani. Component-Oriented Programming with Sharing : Containment is not Ownership. In *Generative Programming and Component Engineering Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 389–404. Springer, 2005. doi :10.1007/11561347_26.

[8] Tom Hirschowitz. Rigid Mixin Modules. In *Functional and Logic Programming Symposium (FLOPS)*, volume 2998 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2004. doi :10.1007/b96926.

[9] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Call-by-value mixin modules : Reduction semantics, side effects, types. In *Programming Languages and Systems European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2004. doi :10.1007/b96702.

[10] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *PPDP '03 : Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declarative programming PPDP '03*, pages 160–171. ACM, 2003. doi :10.1145/888251.888267.

[11] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *Programming Languages and Systems European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 207–236. Springer, 2002. doi :10.1007/3-540-45927-8.

Autres écrits

[12] Richard Garner, Tom Hirschowitz, and Aurélien Pardon. Graphical Presentations of Symmetric Monoidal Closed Theories, 2008.

[13] André Hirschowitz, Michel Hirschowitz, and Tom Hirschowitz. Topological Observations on Multiplicative Additive Linear Logic, 2008.

[14] Tom Hirschowitz and Aurélien Pardon. Binding bigraphs as symmetric monoidal closed theories, 2008.

[15] Tom Hirschowitz. *Mixin Modules, Modules, and Extended Recursion in a Call-by-Value Setting*. Ph.D. thesis, Université Paris 7, 2003.

[16] Tom Hirschowitz. Mixin modules, modules, and extended recursion in a call-by-value setting (extended version), 2003. Version étendue de [15], avec deux chapitres supplémentaires sur les définitions de types à la ML.

[17] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. On the implementation of recursion in call-by-value functional languages. Technical report, INRIA, 2003.

[18] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. A reduction semantics for call-by-value mixin modules. Technical report, INRIA, 2002.