Le langage qui n'existe pas ...

INRIA Sophia Antipolis, 12 février 2007

Résume de mes activité de recherche

(extrait de mon Habilitation à Diriger les Recherches)

Luigi Liquori, Chargé de Recherche INRIA

Nancy (pending), 2006

Table des matières

Première partie

1990-2006

Chapitre 1

Mes Recherches “zippé” dans un chapitre

1.1 Les calculs à prototypes

Les langages à objets ont acquis une importance prépondérante dans les applications informatiques à grande échelle. Cette utilisation a rendu nécessaire l'étude formelle de ces langages pour à la fois mieux en cerner les caractéristiques fondamentales, mais aussi pour pouvoir définir de nouveaux langages à objets et concurrents, capables de combiner une plus grande expressivité avec une sécurité et une efficacité d'utilisation.

Parmi les langages à objets, les langages à prototypes [55] (ou langages à délégation) constituent un axe de recherche important au cours de mes dernières années. Je me suis intéressé plutôt aux aspects théoriques de ces langages, notamment aux *calculs à objets*, calculs où la création des nouveaux objets est déléguée aux objets eux-mêmes (calculs à délégation ou *delegation-based*). Ces calculs sont très importants pour fournir des sémantiques opérationnelles, dénotationnelles et des systèmes de types sophistiqués aux langages à objets.

Le *Lambda Calcul des Objets (LCO)* défini par Fisher, Honsell et Mitchell [34] à l'université de Stanford, et le *Calcul à Objets (OC)* défini par Abadi et Cardelli [6] au Centre de Recherche Digital de Palo Alto en sont les principaux représentants. En particulier, les aspects de typage et d'implantation des dits calculs ont été développés dans [48, 49, 39, 31, 17, 51, 47, 15, 44, 14, 21].

Dans les calculs à objets purs, les objets sont définis directement à partir d'autres objets, en utilisant ces derniers comme prototypes. Les seules opérations sur les objets sont l'extension d'un objet par une nouvelle variable ou une nouvelle méthode (*object extension*), et la surcharge d'une variable ou d'une méthode (*object overriding*). L'objet modifié hérite de toutes les propriétés du prototype. Plusieurs modèles fonctionnels et impératifs avec différents systèmes de typage ont été présentés ces dernières années [54, 1, 3, 2, 4, 5, 7, 6, 56, 57, 33, 34, 35, 16, 60, 61, 13, 19, 62, 36, 37, 38].

Les calculs LCO et OC conduisent à divers systèmes de types qui empêchent statiquement l'erreur `message-not-found` à l'exécution, qui apparaît quand un objet reçoit un message qui n'est pas présent dans son interface. Les systèmes de types pour LCO et OC sont très puissants : en particulier ils permettent la *mytype method specialization*, *c.à d.* la possibilité de spécialiser les types des méthodes héritées. Avec les calculs à prototypes on peut modéliser des langages comme Self [65], Obliq [22], Kevo [63], Emerald [59], Cecil [23], et Omega [12].

Les langages à prototypes (et ses calculs sous-jacents) peuvent aussi être utilisés en tant que "langages cibles" pour implanter et étudier des propriétés formelles des langages à classes, puisque les classes peuvent être vues comme des objets susceptibles de recevoir le message `new` de création d'un objet.

1.1.1 Contributions majeures

– Dans [48] j'ai étudié une extension au premier ordre du calcul des objets primitifs de Abadi et Cardelli ; dans ce calcul les objets peuvent être modifiés en changeant le corps d'un méthode avec un nouveau corps. Cela est suffisant pour modéliser l'héritage simple. Malheureusement l'extension des objets dynamiques (*c.à d.* à la volée comme dans le lambda calcul de Fisher-Honsell-Mitchell) n'est pas prise en compte. L'extension permet de modifier un comportement d'un objet en lui rajoutant des nouvelles méthodes (les méthodes restantes sont héritées). La notion de sous-typage permet d'utiliser un objet avec une interface (*i.e.* la liste des méthodes) dans un contexte qui attend un autre objet avec une interface plus petite (*i.e.* moins de méthodes). Il est bien connu que extension dynamique et sous-typage sont deux notions importantes dans les langages à objets mais difficilement conciliables. J'ai étendu le calcul des objet primitifs avec une opération d'extension dynamique tout en gardant une notion de sous-typage expressif. Le nouveau système de types à été prouvé correct et très expressif pour modéliser classes et héritage entre classes. La table suivante résume la nouvelle syntaxe et sémantique opérationnelle du calcul des objets étendus.

Syntaxe

$$a, b \quad ::= \quad s \;\mid\; [\mathtt{m}_i = \varsigma(s)b_i]^{i\in I} \;\mid\; a.\mathtt{m} \;\mid\; a.\mathtt{m} := \varsigma(s)b$$

Sémantique small-step

Let $a \stackrel{\text{def}}{=} [\mathtt{m}_i = \varsigma(s)b_i]^{i\in I}$

$$(Select) \qquad a.\mathtt{m}_j \rightsquigarrow b_j\{s/a\} \qquad (j \in I)$$

$$(Update) \quad a.\mathtt{m}_j := \varsigma(s)b \rightsquigarrow [\mathtt{m}_i = \varsigma(s)b_i, \mathtt{m}_j = \varsigma(s)b]^{i\in I\setminus\{j\}} \quad (j \in I)$$

$$(Extend) \quad a.\mathtt{m}_j := \varsigma(s)b \rightsquigarrow [\mathtt{m}_i = \varsigma(s)b_i, \mathtt{m}_j = \varsigma(s)b]^{i\in I} \qquad (j \notin I)$$

L'extension demande l'introduction d'un nouveau type, le *type diamant* qui a la forme $[\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j\in J}^{i\in I}$. Intuitivement, les méthodes à gauche du diamant (*partie interface*) sont réellement présentes dans l'objet, tandis que les méthodes à droite du diamant (*partie sous-typable*) ont été oubliées par sous-typage ou seront ajoutées dans l'objet qui "habite" ce type. La table suivant retient les règles les plus importantes du nouveau système de type pour le calcul des objets étendus.

$$\frac{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j\in J}^{i\in I+K}}{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j\in J}^{i\in I+K} <: [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j\in J+K}^{i\in I}}\ (\text{Shift}_\diamond)$$

$$\frac{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j\in J+K}^{i\in I}}{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j\in J}^{i\in I} <: [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j\in J+K}^{i\in I}}\ (\text{Extend}_\diamond)$$

$$\frac{\Gamma, s_i : \tau_i \vdash b_i : \sigma_i \qquad \forall i \in I \qquad H_i \subseteq I}{\Gamma \vdash [\mathtt{m}_i = \varsigma(s_i{:}\tau_i) b_i]^{i\in I} : [\mathtt{m}_i : \sigma_i \diamond]^{i\in I}}\ (\text{Object})$$

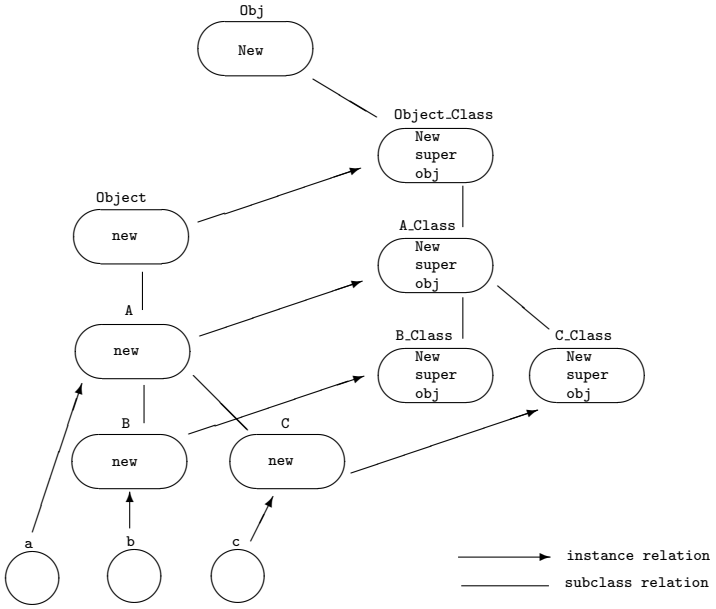
$$(\text{Let } \tau_k \stackrel{\text{def}}{=} [\mathtt{m}_h : \sigma_h]^{h\in H}).$$

$$\frac{\Gamma \vdash a : [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j\in J}^{i\in I} \qquad \Gamma, s_k : \tau_k \vdash b : \sigma_k \qquad H \subseteq I \qquad k \in J}{\Gamma \vdash a.\mathtt{m}_k := \varsigma(s_k{:}\tau_k) b : [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j\in J-\{k\}}^{i\in I+\{k\}}}\ (\text{Ext})$$

Intuitivement :

- La règle ($Shift_\diamond$) dit qu'on peux "cacher" une méthode qui appartient à la partie interface en le déplaçant à droite du diamant ; si cette méthode est ajoutée par la suite, son type sera donc mémorisé ;
- La règle ($Extend_\diamond$) dit que un objet avec peu de méthodes dans la partie sous-typable peut être utilisée dans tout contexte qui attend un objet typable avec un type avec plus de méthodes dans la partie sous-typable. Cette règle permet de ajouter de nouvelles méthodes jamais introduits ;
- La règle ($Object$) est la même que celle du calcul original d'Abadi et Cardelli si on dit que $H_i \stackrel{set}{=} I$, et $[\mathtt{m}_i : \sigma_i \diamond]^{i\in I} \stackrel{\text{def}}{=} [\mathtt{m}_i : \sigma_i]^{i\in I}$;
- La règle (Ext) est le cœur de l'extension : tout d'abord on type l'objet a avec un type diamant ; puis, on type le corps de la méthode $\mathtt{m}$ avec un type σ_i, et enfin, on vérifie que $\mathtt{m} : \sigma_i$ est présent dans la partie sous-typable du type diamant de a. Cela permet de traiter les cas ou la méthode $\mathtt{m}$ est fraîche ou appartient déjà à l'objet a mais avait été "cachée" par sous-typage.

Avec le calcul des objets étendus on peut coder une hierarchie de classes et metaclasses *à la*SmallTalk, comme montré dans la table suivante (le codage complet se trouve dans le papier [48]) :



– Dans [39], avec Furio Honsell et Pietro di Gianantonio (Un. Udine) on a travaillé sur un nouveau système de typage pour le λ-calcul des objets de Fisher & Honsell & Mitchell (LCO) permettant de typer la self-extension (*i.e.* l'objet s'auto éteint suite à la réception d'un message). La syntaxe, une nouvelle sémantique opérationnelle et un nouveau système de types ont été présentés. La nouvelle sémantique est plus efficace dans la recherche de la méthode de l'object qu'on appelle ; elle est fondée sur les R-réductions de Klop.

Syntaxe

$$
\begin{aligned}
M, N \ ::= \ & c \mid s \mid \lambda s.M \mid M\,N \mid \\
& \langle\,\rangle \mid \langle M \longleftarrow \mathtt{m} = N \rangle \mid \langle M \longleftarrow\!\!+ \mathtt{m} = N \rangle \mid \\
& M \Leftarrow \mathtt{m} \mid Sel(M, \mathtt{m}, N)
\end{aligned}
$$

Sémantique small-step $(\longleftarrow\!\!* \;=\; \longleftarrow \text{ ou } \longleftarrow\!\!+\,)$

$$(Beta) \qquad (\lambda s.M)\,N \;\rightsquigarrow\; M[s/N]$$

$$(Sel) \qquad M \Leftarrow \mathtt{m} \;\rightsquigarrow\; Sel(M, \mathtt{m}, M)$$

$$(Next) \quad Sel(\langle M \longleftarrow\!\!* \mathtt{n} = N\rangle, \mathtt{m}, P) \;\rightsquigarrow\; Sel(M, \mathtt{m}, P) \qquad (\mathtt{m} \neq \mathtt{n})$$

$$(Succ) \quad Sel(\langle M \longleftarrow\!\!* \mathtt{n} = N\rangle, \mathtt{n}, P) \;\rightsquigarrow\; N\,P$$

La relation d'entrée/sortie est la suivante pour des valeurs v qui sont des objets ou des lambda-abstractions :

$$\frac{}{v \Downarrow v}\,(\text{Big}-\text{Value})$$

$$\frac{M \Downarrow \lambda x.M' \qquad [N/x]M' \Downarrow v}{M\,N \Downarrow v}\,(\text{Big}-\text{Beta})$$

$$\frac{\mathtt{e} \Downarrow obj \qquad Sel(obj, \mathtt{m}, obj) \Downarrow v}{M \Leftarrow \mathtt{m} \Downarrow v}\,(\text{Big}-\text{Select})$$

$$\frac{N\,obj \Downarrow v}{Sel(\langle M \longleftarrow\!\!* \mathtt{m} = M\rangle, \mathtt{m}, obj) \Downarrow v}\,(\text{Big}-\text{Success})$$

$$\frac{M \Downarrow obj' \qquad Sel(obj', \mathtt{m}, obj) \Downarrow v \qquad \mathtt{m} \neq \mathtt{n}}{Sel(\langle M \longleftarrow\!\!* \mathtt{n} = N\rangle, \mathtt{m}, obj) \Downarrow v}\,(\text{Big}-\text{Next})$$

Comme exemples de l'expressivité on considère le point suivant :

$$\begin{aligned} point \stackrel{\text{def}}{=}\; & \langle add\_set\_col = \lambda self.\lambda v.\langle self \longleftarrow\!\!* col = v\rangle, \\ & x = \lambda self.1\rangle \end{aligned}$$

Si on envoie le message *add_set_col* avec l'argument *red*, *i.e.* $point \Leftarrow add\_set\_col(red)$ à *point* on obtiendra un nouveau objet où le champ couleur à été ajouté, *c.à d.* :

$$\begin{aligned} col\_point \stackrel{\text{def}}{=}\; & \langle add\_set\_col = \lambda Self.\lambda v.\langle self \longleftarrow\!\!* col = \lambda Self'.v\rangle, \\ & x = \lambda Self.1, \\ & col = \lambda Self'.red\rangle \end{aligned}$$

Si on envoie pour une deuxième fois à *col_point* le message *add_set_col* avec l'argument *black*, puisque le champ couleur est déjà présent, le résultat sera de rédefinir la couleur

de l'objet en *black*, *i.e.* :

$$\begin{aligned} col\_point \Leftarrow add\_set\_col(black) \;\mapsto\; \langle add\_set\_col = \lambda Self.\lambda v. \\ \langle self \longleftarrow\!\!* col = \lambda Self'.v\rangle, \\ x = \lambda Self.1, \\ col = \lambda Self'.black\rangle \end{aligned}$$

– Dans [14] avec Mariangiola Dezani-Ciancaglini, Viviana Bono (Un. Turin) et Michele Bugliesi (Un. Venise), on a essayé d'étendre le système de typage du lambda calcul des objets de avec la possibilité de typer des *objets incomplets c.à d.* des objets qui n'ont pas toutes les méthodes encore implémentées. Ceci rappelle le *classes abstraites* de C++. Par exemple on considère l'objet `abs` qui renvoie le logarithme du champ `x` si positif ou appelle une exception si négatif.

$$\begin{aligned} \langle\langle\ \mathtt{x} = \lambda \mathtt{self}.\ 0\rangle \longleftarrow\!\!* \mathtt{safe\_log} = \lambda \mathtt{self}.\ & \mathtt{if}\ (\mathtt{self} \Leftarrow \mathtt{x}) \geq 0 \\ & \mathtt{then}\ \mathtt{log}\,(\mathtt{self} \Leftarrow \mathtt{x}) \\ & \mathtt{else}\ \mathtt{self} \Leftarrow \mathtt{handle\_ex}\ \rangle. \end{aligned}$$

L'objet `abs` sera typable avec le type suivant :

$$\varepsilon \vdash \mathtt{abs} : \mathtt{class}\, t.\langle\!\langle \mathtt{x} : int, \mathtt{safe\_log} : real_{\{\mathtt{handle\_ex}\}}\rangle\!\rangle \circ \langle\!\langle \mathtt{handle\_ex} : real\rangle\!\rangle.$$

Intuitivement, dans les méthodes déclarées à gauche du $\circ$ sont réellement présentes dans l'objet `abs` tandis que les méthodes déclarés a droite du *circ* ne sont pas encore implémentées dans l'objet. Dans un certain sens ces méthodes jouent le rôle des méthodes virtuelles pures *à la* C++ Tout appel a `abs` $\Leftarrow$ `safe_log` ne typera pas car la méthode `safe_log` dépend de la méthode `handle_ex`, qui n'est pas encore présente dans l'objet `abs`. Ce qui est intéressant dans ce travail est le fait que l' objet `abs` peut etre utilisé pour construire de nouveaux objets, comme par exemple l'objet `handler` :

$$\mathtt{handler} \stackrel{\triangle}{=} \langle \mathtt{abs} \longleftarrow\!\!* \mathtt{handle\_ex} = \lambda \mathtt{self}.-1\rangle$$

qui implémente la méthode `handle_ex`. L'objet `handler` sera typable avec le type :

$$\mathtt{handler} : \mathtt{class}\, t.\langle\!\langle \mathtt{x} : int, \mathtt{safe\_log} : real_{\{\mathtt{handle\_ex}\}}, \mathtt{handle\_ex} : real\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle,$$

Avec ce typage, l'expression `handler` $\Leftarrow$ `safe_log` sera typable car le jugement de type `handler` $\Leftarrow$ `safe_log` : *real* sera dérivable.

– Avec Michele Bugliesi (Un. Venice) et Giorgio Delzanno (Un. Génes) et Maurizio Martelli (Un. Pise) on a étudié un calcul à prototypes qui se base sur la logique linéaire de Girard. Ce calcul peut être vu comme la partie logique correspondante aux calculs à objets OC et LCO. Mon principal intérêt a été celui d'étudier les bases logiques de ce calcul, en montrant que des concepts à objets comme l'héritage, la liaison dynamique, l'envoi de message, la surcharge de méthodes et l'extension des objets peuvent être facilement pris en compte dans un langage logique inspiré de la logique linéaire d'une façon naturelle [20, 21]. La syntaxe du calcul logique à objets est la suivante :

Objets	$O ::= s, x, y, \ldots$	variables
	$\mid$ `obj`$[\,]$	l'objet vide
	$\mid O.m := \forall s\, M$	ajoute/redéf. des méth.
Def. de Méthodes	$M ::= A \mapsto O$ `if` B	définition conditionnelle
	$\mid A \mapsto O$	définition atomique
	$\mid \forall \vec{x}\, M$	déf. avec quantificateurs
Corps des méthodes	$B ::= O.m\ A \mapsto O$	envoi de message
	$\mid B, B$	conjonction
Liste d'argument	$A ::= (O_1, \ldots, O_n)$	$n \geq 0$ arguments
Requête	$Q ::= B \mid \exists x.Q$	

Et un point diagonal pourrait etre exprimé en la façon suivante :

$$diag\_point \stackrel{\text{def}}{=} \texttt{obj}[x = \forall s.(\_) \mapsto 1, y = \forall s. \forall v.(\_) \mapsto v \texttt{ if } s.x\ (\_) \mapsto v]$$

1.2 Les calculs à classes

Les langages à classe sont quasi universellement considérés comme intrinsèquement impératifs : en fait ils ont une notion d'état, *c.à d.* de variables propres à chaque instance d'objet. Néanmoins :

- une grande partie de la littérature scientifique sur la théorie des langages à objets a été développée au travers de petits langages à objets fonctionnels purs (voir par exemple, à ce sujet, le papier sur *Featherweight Java* de A. Igarashi, B. Pierce et P. Wadler [42]) ;
- des concepts comme l'héritage, la liaison dynamique, l'envoi de message, la surcharge de méthodes et l'extension dynamique des objets ne sont pas du tout en contradiction avec le paradigme fonctionnel lui même.

La veine de recherche que j'ai suivie dans ces derniers années est d'étendre Featherweight Java avec une notion d'heritage multiple basée sur le mechanisme de traits.

1.2.1 Contribution majeures

Featherweight-Trait Java une extension de FJ basée sur les traits. Avec Arnaud Spiwack, ENS Cachan, dans le contexte des langages typés statiquement et basés sur les classes, nous avons étudié un mécanisme pour étendre les classes par la composition de *traits*. Construire les classes en composant des ensembles de méthodes est une technique connue pour implémenter les langages basés sur les objets ou les classes avec héritage simple. Elle a été étudiée, seulement récemment, comme méchanisme de première classe disponible à l'utilisateur dans un contexte non typé.

Cette veine de recherche nous a mené à présenter Featherweight-Trait Java (FTJ), une extension conservative de Featherweight Java (FJ), un calcul simple et léger basé sur les classes auquel on a rajouté des traits typés statiquement. Dans FTJ, les classes peuvent être construites en utilisant les traits comme des briques basiques de comportements : les traits contiennent seulement des comportements, et pas d'états. Les conflits entre les méthodes doivent être résolues explicitement par l'utilisateur soit (1) en renommant ou en excluant la méthode, soit (2) en redéfinissant explicitement les méthodes concernées dans la classes. Un

certain accent a été mis sur les "conflits en diamants", une question habituelle de l'héritage multiple.

Ceci rend une sorte d'héritage multiple non seulement possible, mais facile. Nous avons présenté une nouvelle sémantique opérationnelle avec un nouvel algorithme de recherche, et un nouveau système de typage qui garantit que l'évaluation ne soulève pas une erreur "message-pas-compris", et que l'interpréteur ne se bloque pas. Le calcul proposé semble être un bon point de départ pour une analyse mathématique rigoureuse de langages typés basés sur les traits, et un nouveau paradigm pour la programmation orientée trait. La syntaxe de FTJ est la suivante :

Syntax

CL	::= class C extends C $\{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\ \overline{\texttt{TA}}\}$	Class Declarations
TL	::= trait T is $\{\overline{\texttt{M}}; \overline{\texttt{TA}}\}$	Trait Declarations
TA	::= T | TA with$\{\texttt{m@m}\}$ | TA minus $\{\texttt{m}\}$	Trait Alterations
K	::= $\texttt{C}(\overline{\texttt{C}}\ \overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{f}}); \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};\}$	Constructors
M	::= $\texttt{C}\ \texttt{m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\{\texttt{return e};\}$	Methods
e	::= x | e.f | $\texttt{e.m}(\overline{\texttt{e}})$ | new $\texttt{C}(\overline{\texttt{e}})$ | (C)e	Expressions

L'algorithme de "lookup" implement le trois régles suivants.

1. La méthode définit dans une classe a la priorité sur la méthode définit dans un trait qui est importé dans la classe ;
2. La méthode définit dans un trait a la priorité sur la méthode qui est importé dans le trait ;
3. La méthode définit dans un trait et importé par une classe a la priorité sur la méthode définit dans la superclasse.

Quelque exemple suit.

```
trait T1 is {String p(){return ``hello'';}}
trait T2 is {String p(){return ``world'';}}
trait T3 is {String m(){return (...this.p()...);}
                                                    p is a required method
             String p(){return ``hello world'';}
                                                    p is an overridden method
             String n(){return (...this.q()...);}
                                                    q is a required method
             T1 T2}
          Trait T3 imports traits T1 and T2, override p, and q is still a required method
trait T4 is {Object p(){return(...this.r()...);}
                                                    r is a required method
```

```
trait T5 is {Object q(){return(...this.s()...);}
                                                    s is a required method
trait T6 is {Object m(){return(...this.p()...);}
                                                    p is a required method
             Object n(){return(...this.q()...);}
                                                    q is a required method
             T4 T5}
                 Trait T6 imports traits T4 and T5 and r and s are still required methods
```

Résolution de conflits L'héritage via trait peut engendrer des conflits ; par exemple, une classe C peut importer deux traits T1 et T2 qui ont une même méthode p avec corps différents. Dans ce cas, le conflit doit être résolu *à la main*. Une fois que un conflit a été détecté (par le compilateur) on a deux façons de résoudre le conflit :

1. **Ré-écrire un nouveau méthode p dans la classe.** L'algorithme de liaison dynamique sélectionnera la méthode définie dans la classe. Par exemple :

   ```
   class C extends Object
   {...;                                          instance vars
    ...                                           constructor
    D p(...){...}                                 new behavior for p, the winner
    T1 T2}                                        each trait defines a (different) behavior for p
   ```

2. **Renommer (*Aliasing*) les méthodes dans les traits et ré-écrire un nouveau méthode p dans la classe.** La méthode p est renommé dans T1 et T2 et une nouvelle méthode p sera redéfinie dans la classe C (qui pourra réutiliser les méthodes renommées p_of_T1 et p_of_T2 qui ne sont plus en conflit). Par exemple :

   ```
   class C extends Object
   {...;                                          instance vars
    ...                                           constructor
    D  p(...){...}                                new winner behavior for p, it may use p_of_T1/2
    T1 with {p@p_of_T1}                           T1 aliases p with p_of_T1
    T2 with {p@p_of_T2}}                          T2 aliases p with p_of_T2
   ```

3. **Exclure la méthode p dans un trait.** La méthode p dans le trait T1 ou T2 est éliminé : ceci résolue le conflit en faveur d'un trait. Par exemple :

   ```
   class C extends Object
   {...;                                          instance vars
    ...                                           constructor
    T1                                            contains the winner method p
    T2 minus {p}}                                 method p is now hidden
   ```

1.3 La théorie de types dépendants

Les théories typées, grâce à l'isomorphisme de Curry-Howard, [40] fournissent une base théorique aux assistants à la preuve. Suivant cet isomorphisme, une propriété à démontrer est spécifiée par un type et une démonstration de cette propriété devient un λ-terme (*c.à d.* un programme fonctionnel) ayant ce type. On voit que la puissance d'expression de la théorie

choisie a une grande influence sur le niveau d'abstraction avec lequel l'utilisateur pourra exprimer la propriété à prouver, comme sur la facilité (ou la difficulté !) qu'il aura à faire sa preuve. Ces dernières années ont vu différentes tentatives pour enrichir les théories typées connues, par l'ajout de modules, de sous-typage, de types inductifs, de types quotients et même de types objets. L'un des mes axes de recherche est la recherche de nouvelles théories typées permettant la récursion sur des termes de type fonctionnel.

Les assistants à la preuve basés sur la thèorie des types dépendant sont très utilisés ces dernières années pour essayer de démontrer des propriétés très difficiles comme, par exemple, la normalisation forte d'un calcul ou la conservation de types pour un langage de programmation. Je me suis s'intéressé aussi à la production de *logiciels sûrs*. Dans ce domaine, les systèmes basés sur une théorie typée vérifiant l'isomorphisme de Curry-Howard sont non seulement *a priori* les plus fiables (puisque leurs fondements sont connus), mais aussi les plus puissants, en terme d'expressivité. Dans le cas où l'on peut extraire un programme à partir de sa spécification, la preuve de correction de programme devient *programmation certifiée*. La spécification d'un langage de programmation se prête bien à être formalisée à l'aide de propriétés mathématiques directement transposables dans les démonstrateurs de théorèmes. La preuve de correction d'un compilateur (qui est lui-même un logiciel) représente un exemple typique de l'utilité des démonstrateurs de théorèmes.

Parmi les assistants à la preuve, l'un des plus puissant semble être le système Coq : avec son système de types basé sur les types (co-)inductifs, il permet la spécification des langages (exprimés à l'aide des grammaires inductives) et de leurs systèmes de types. Avec Coq, ont été développées récemment beaucoup de bibliothèques : la preuve des propriétés des programmes impératifs, la modélisation de la *scope extrusion* dans le π-calcul, ainsi que la normalisation forte d'une restriction de Coq lui-même (voir le papier "*Coq in Coq*" de B. Barras [10], une sorte de mini-bootstrap du système lui même), etc.

Pourquoi étudier ces systèmes ? Cette veine de recherche théorique est-elle importante ? Y-a-t-il des débouchés à la concrétisation de ces mathématiques sous forme de logiciels d'assistance à la preuve semi-automatique ?

> *Les logiciels sont de plus en plus complexes. Hier encore, un logiciel représentait quelques milliers de lignes de code, tandis qu'aujourd'hui de façon courante ces produits en nécessitent plusieurs millions, ce qui entraîne une quantité plus importante d'erreurs de programmation. Une étude menée par le National Institute of Standards and Technology évalue à 60 milliards de dollars par an le coût des bogues informatiques, dont 22 milliards de dollars pourraient être évités si l'industrie informatique améliorait ses équipements de test des logiciels. Dans ce secteur qui emploie 700 000 ingénieurs et dont le chiffre d'affaires atteint 180 milliards de dollars, les développeurs consacrent environ 80% de leurs dépenses à identifier et à corriger des erreurs de programmation, mais cela ne suffit pas à empêcher la multiplication des bogues [67].*

Dans le domaine du logiciel certifié, on distingue en général la validation d'outils généraux pour les langages de programmation (tels que des interpréteurs, ou des compilateurs) de la validation des programmes eux-mêmes.

Dans le premier cas, on parle de *preuve de propriétés de langages*. Deux exemples typiques dans ce domaine sont la preuve de conservation de types d'un langage lors de son exécution, et la preuve de correction d'un compilateur. La première propriété assure une certaine cohérence entre le système de vérification de type et les règles d'évaluation d'un langage. Ce type de

preuve, essentiel pour les concepteurs du langage, donne par ailleurs au programmeur une certaine confiance dans les outils qu'il utilise.

La *preuve de correction de programmes* consiste généralement à prouver certaines propriétés d'un programme, par exemple la validité d'un invariant. Cependant, dans le cas idéal où l'on peut extraire un programme à partir de sa spécification, la preuve de correction de programmes devient *programmation certifiée*. À l'intersection des deux domaines, on trouve la certification d'un interpréteur, ou d'un compilateur.

Plus généralement, les compétences sur les preuves formelles intéressent potentiellement l'ensemble des domaines industriels où la présence d'erreurs même minimes dans les programmes peut avoir des conséquences graves : danger pour la vie humaine comme dans l'énergie nucléaire ou les transports (avions, automobiles), la chirurgie assistée par ordinateur, ou simplement coût prohibitif comme dans le commerce électronique et les réseaux de télécommunication, l'aérospatial, les *SoC, i.e. Systems on Chips*, les cartes à puces ou bien dans les domaines où les enjeux financiers sont importants comme le domaine bancaire. Tous ces domaines ont besoin de *logiciels certifiés*.

De nombreuses équipes de recherche mettent au point des techniques qui permettent de développer des logiciels validés vis-à-vis de spécifications, ce qui permettrait "théoriquement" de réduire le risque d'erreur à zéro (en "pratique" des erreurs peuvent également se glisser dans les spécifications). Ces techniques sont plus ou moins coûteuses et plus ou moins générales. Avec le logiciel Coq, fondé sur une théorie typée du Calcul des Constructions (co-)Inductives, on se trouve à un bout du spectre : on peut aborder des problèmes variés et complexes mais avec un coût de développement plus important que d'habitude[1]. Il est donc intéressant de diminuer ce coût, d'une part par la recherche de théories typées et de méthodes de formalisation permettant des spécifications et des preuves plus concises, d'autre part par le développement de bibliothèques d'exemples.

1.3.1 Contributions majeures

– **Systèmes de types dépendants pour le lambda calcul pur** H. Barendregt [9] a donné une représentation compacte d'une classe des systèmes à la Church pour le lambda calcul et il les a placés sur les sommets d'un "cube". J'ai travaillé avec Simona Ronchi (Un. Turin), Pawel Urzyczyn (Un. Varsovie) et Steffen van Bakel (Imperial College, Londres) sur des systèmes d'assignation de types pour le lambda calcul à la Curry. En particulier, je me suis intéressé à l'étude des systèmes d'assignation pour le lambda calcul avec des types dépendant des termes [66].
La partie sans types dépendants contient des systèmes connus dans la littérature (par exemple le système du second ordre de Leivant [45]). Les systèmes d'assignation de types avec types dépendant des termes (à ma connaissance des systèmes originaux) [66] ont de bonnes propriétés opératoires comme *Subject Reduction* et *Strong Normalisation*.
Les relations qui existent entre le cube de Barendregt et le cube à la Curry sont très intéressantes : en particulier, j'ai démontré que, pour quelques systèmes d'assignation de types du cube à la Curry, un jugement ne peut pas être obtenu comme *type erasure* d'un jugement correspondant dans le cube de Barendregt.
Les types dépendants sont intéressants, du point de vue logique et aussi du point de vue des langages de programmation : en fait la possibilité de définir des structures de

[1] Les techniques de *Model Checking* sont à l'autre bout du spectre.

listes paramétrées par le nombre de leurs éléments s'exprime avec le type dépendant $\Pi n : int.\mathsf{list}(n)$, où $\mathsf{list}(\_)$ est une constante de liste paramétrique.
La syntaxe du cube *à la* Curry est la suivante :

$$\begin{aligned} M &::= x \mid \lambda x.M \mid MM \\ \phi &::= \alpha \mid \Pi x{:}\phi.\phi \mid \Pi\alpha{:}K.\phi \mid \lambda x{:}\phi.\phi \mid \lambda\alpha{:}K.\phi \mid \phi\phi \mid \phi M \\ K &::= * \mid \Pi x{:}\phi.K \mid \Pi\alpha{:}K.K \end{aligned}$$

Elle a été obtenue a partir de la syntaxe du cube de Barendregt en appliquant la fonction d'effacement $E : \mathcal{T}_t \to \mathcal{T}_u$ qui est définie inductivement comme suit :
– On Λ_t.

$$\begin{aligned} E(x) &= x, \\ E(MN) &= E(M)E(N), \\ E(M\phi) &= E(M), \\ E(\lambda x{:}\phi.M) &= \lambda x.E(M), \\ E(\lambda\alpha{:}K.M) &= E(M). \end{aligned}$$

– On $Cons_t$.

$$\begin{aligned} E(\alpha) &= \alpha, \\ E(\Pi x{:}\phi.\psi) &= \Pi x{:}E(\phi).E(\psi), \\ E(\Pi\alpha{:}K.\psi) &= \Pi\alpha{:}E(K).E(\psi), \\ E(\lambda x{:}\phi.\psi) &= \lambda x{:}E(\phi).E(\psi), \\ E(\lambda\alpha{:}K.\psi) &= \lambda\alpha{:}E(K).E(\psi), \\ E(\phi\psi) &= E(\phi)E(\psi), \\ E(\phi M) &= E(\phi)E(M). \end{aligned}$$

– On $Kind_t$.

$$\begin{aligned} E(*) &= *, \\ E(\Pi x{:}\phi.K) &= \Pi x{:}E(\phi).E(K), \\ E(\Pi\alpha{:}K_1.K_2) &= \Pi\alpha{:}E(K_1).E(K_2). \end{aligned}$$

Une observation importante est que dans cette stratification, la seule grammaire *à la* curry est la syntaxe des lambda-termes ; les autres (constructeurs et *kinds*) sont encore *à la* Church.
Les règles d'inférences de ce cube sont données en termes des jugements de type de la forme $\Gamma \vdash A : B$, où Γ est un contexte et $A : B$ représente une *affectation de type*.
– *Common Rules*

$$(Proj)\quad \frac{\Gamma \vdash A : s \quad a \notin \mathcal{D}om(\Gamma)}{\Gamma, a{:}A \vdash a : A} \qquad (Weak)\quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s \quad c \notin \mathcal{D}om(\Gamma)}{\Gamma, c{:}C \vdash A : B}$$

$$(Conv)\quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s \quad B =_\beta C}{\Gamma \vdash A : C}$$

– *λ-Term Rules*

$$(I)\quad \frac{\Gamma, x{:}\phi \vdash M : \psi}{\Gamma \vdash \lambda x.M : \Pi x{:}\phi.\psi} \qquad (E)\quad \frac{\Gamma \vdash M : \Pi x{:}\phi.\psi \quad \Gamma \vdash N : \phi}{\Gamma \vdash MN : \psi[N/x]}$$

$$(I_K)\quad \frac{\Gamma, \alpha{:}K \vdash M : \phi}{\Gamma \vdash M : \Pi\alpha{:}K.\phi} \qquad (E_K)\quad \frac{\Gamma \vdash M : \Pi\alpha{:}K.\phi \quad \Gamma \vdash \psi : K}{\Gamma \vdash M : \phi[\psi/\alpha]}$$

– *Constructor Rules*

$$(C\text{–}I_C)\quad \frac{\Gamma, x{:}\phi \vdash \psi : K}{\Gamma \vdash \lambda x{:}\phi.\psi : \Pi x{:}\phi.K} \qquad (C\text{–}E_C)\quad \frac{\Gamma \vdash \psi : \Pi x{:}\phi.K \quad \Gamma \vdash M : \phi}{\Gamma \vdash \psi M : K[M/x]}$$

$$(C\text{–}I_K)\quad \frac{\Gamma, \alpha{:}K_1 \vdash \psi : K_2}{\Gamma \vdash \lambda\alpha{:}K_1.\psi : \Pi\alpha{:}K_1.K_2} \qquad (C\text{–}E_K)\quad \frac{\Gamma \vdash \phi : \Pi\alpha{:}K_1.K_2 \quad \Gamma \vdash \psi : K_1}{\Gamma \vdash \phi\psi : K_2[\psi/\alpha]}$$

$$(C\text{–}F_C)\quad \frac{\Gamma, x{:}\phi \vdash \psi : *}{\Gamma \vdash \Pi x{:}\phi.\psi : *} \qquad (C\text{–}F_K)\quad \frac{\Gamma, \alpha{:}K \vdash \phi : *}{\Gamma \vdash \Pi\alpha{:}K.\phi : *}$$

– *Kind Rules*

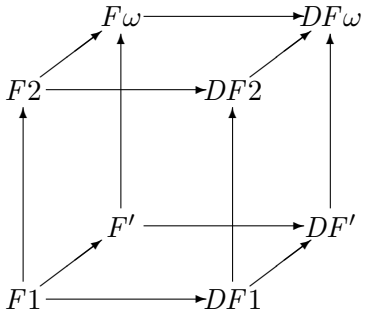
$$(Axiom)\quad \frac{}{<> \vdash * : \Box} \qquad (K\text{–}F_C)\quad \frac{\Gamma, x{:}\phi \vdash K : \Box}{\Gamma \vdash \Pi x{:}\phi.K : \Box}$$

$$(K\text{–}F_K)\quad \frac{\Gamma, \alpha{:}K_1 \vdash K_2 : \Box}{\Gamma \vdash \Pi\alpha{:}K_1.K_2 : \Box}$$

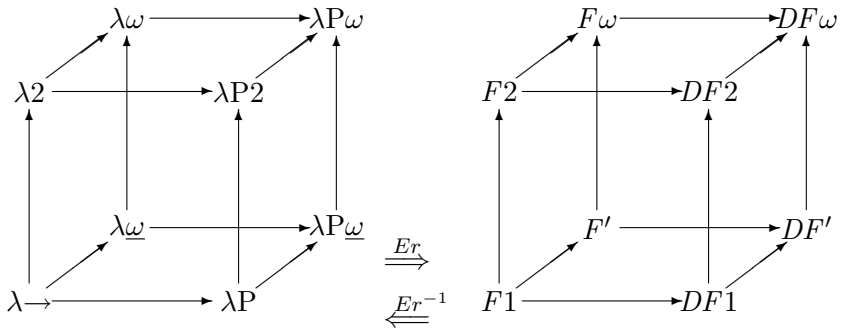
Le point délicat de ce système de type est la présence de règles qui ne sont pas dirigées par la syntaxe, *c.à d.* le sujet de l'affectation de type est toujours la même. Ce règles sont (I_K), (E_K), $(Weak)$ et $(Conv)$.
En utilisant des sous-ensembles précis de ces règles, on a pu distinguer huit systèmes de types qui sont bien identifiés dans le TAS-cube.

$$\begin{aligned}
F1 &= \textit{Base-Rules},\\
F' &= F1 \cup \textit{Higher-Order},\\
F2 &= F1 \cup \textit{Polymorphism},\\
F\omega &= F1 \cup \textit{Higher-Order} \cup \textit{Polymorphism},\\
DF1 &= F1 \cup \textit{Dependencies},\\
DF' &= F1 \cup \textit{Dependencies} \cup \textit{Higher-Order},\\
DF2 &= F1 \cup \textit{Dependencies} \cup \textit{Polymorphism},\\
DF\omega &= F1 \cup \textit{Dependencies} \cup \textit{Higher-Order} \cup \textit{Polymorphism}.
\end{aligned}$$

Comme pour le fameux Cube de Barendregt, pour chaque ensemble de règles, S, l'expression $\Gamma \vdash_S A : B$ indiquera que $\Gamma \vdash A : B$ peut être dérivable en utilisant seulement le règles dans S. Les huit systèmes peuvent être disposées sur le noeuds du cube suivant :



Les trois dimensions dans ce cube correspondent, comme pour le cube de Barendregt, à l'introduction de types polymorphes, d'ordre supérieur et dépendant. En particulier les systèmes dans la partie droite du cube sont nouveaux et leur étude représente une importante contibution originale. La relation entre le cube de Barendregt et le nouveau cube est la suivante :



– **Systèmes des types dépendant pour un *logical framework* générique.** Avec Furio Honsell et Marina Lenisa (Univ. Udine) nous avon étudié un *Cadre Logique Générique* appelé GLF, qui permet de définir des cadres logiques. Il est basé sur la discipline des types dépendants, dans le style du fameux Cadre Logique d'Edinburgh LF. Le cadre GLF est caractérisé par une forme généralisée de lambda abstraction pour laquelle la β-réduction s'applique à condition que l'argument puisse satisfaire un prédicat logique et en produisant une substitution. Le système de typage mémorise le fait qu'il y a une constrainte à satisfaire pour que la réduction s'applique. Le cadre GLF capture, comme instance, LF ainsi qu'une classe étendue de lambda calculs avec contraintes, et des lambda calculs bien connus comme, par exemple, le lambda calcul avec appel par valeur de Plotkin, ou le lambda calcul avec motifs. Mais il ouvre aussi sur un spectre plus large de nouveaux calculs.
Nous avous étudié les propriétés metathéoriques du calcul sous-jacent à GLF et illustré son pouvoir expressif. En particulier nous nous concentrons sur deux instances intéressantes de GLF. La première est le Cadre Logique avec Motifs (PLF) dans lequel une application se déclenche à travers le filtrage de motifs dans le style de Cirstea, Kirchner et Liquori. Le second est le Lambda Calcul Clos (CLF) qui en outre de la β-réduction standard, possède une réduction qui se déclenche seulement si l'argument est une expression *close*. Pour ces deux instances de GLF, nous étudions leurs metaproprietés standard telles que Church-Rosser, la préservation du type par réduction et la normalisation forte.
Le cadre GLF est particulièrement adapté, en tant que métalangage, pour coder des logiques de réécriture et des systèmes logiques, dans lesquelles les règles imposent aux

termes de preuve de respecter des contraintes syntaxiques particulières, telles que des logiques avec *règles de preuve* adjointes aux *règles de dérivations*, comme on peut les trouver, par exemple, en logique modale.

1.4 La réécriture

Le Rho-calcul est un calcul qui intègre les propriétés complémentaires de la réécriture du premier ordre et du Lambda-calcul ainsi que des caractéristiques permettant d'exprimer le non-déterminisme. Ce calcul est suffisamment puissant pour décrire non seulement la réécriture avec des règles conditionnelles mais aussi leur contrôle. Ainsi, le Rho-calcul nous permet de représenter les termes et les réductions du Lambda-calcul et de la *réécriture conditionnelle*. Le Rho-calcul est un bon candidat pour donner une sémantique à l'application des règles du langage ELAN [18], un cadre logique dont le noyau est la logique de réécriture étendue avec la notion de stratégies.

En partant de la représentation des règles de réécriture conditionnelles, le Rho-calcul peut être employé pour donner une sémantique à l'application des règles du langage ELAN [18], un cadre logique dont le noyau est la logique de réécriture étendue avec la notion de stratégies.

Les stratégies ELAN sont, dans la plupart des cas, représentées directement dans le Rho-calcul. En effet, l'expressivité du Rho-calcul nous permet de représenter explicitement le mécanisme d'évaluation ELAN, basé sur l'utilisation d'une stratégie de normalisation *innermost* pour les règles non-nommées et des stratégies définies pour les règles nommées. Nous pouvons donc représenter un programme ELAN par un ρ-terme approprié ; le Rho-calcul fournit donc une sémantique opérationnelle complète au langage ELAN.

1.4.1 Contribution majeures

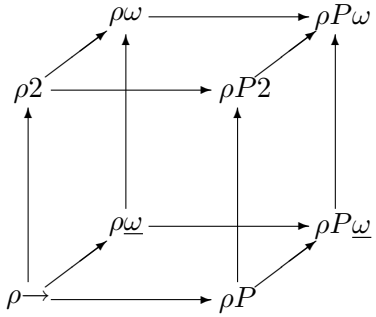
– **"Réécriture" du calcul de réécriture.** Dans [26] je me suis intéressé avec Claude Kirchner et Horatiu Cirstea de l'équipe Protheo (LORIA), à définir une nouvelle syntaxe et une nouvelle sémantique opérationnelle et de systèmes de typage pour le Rho-calcul par rapport à la définition originale présenté dans la thèse de Horatiu en 1998. La nouvelle syntaxe du Rho-calcul est très simple :

$$\begin{array}{lll} t ::= \ ::= \ a \mid X \mid t{\rightarrow}t \mid t@t \mid & & \text{term simple} \\ \\ null \mid t \wr t & & \text{séquences} \end{array}$$

où a dénote une constante, X dénote une variable, $t{\rightarrow}t$ une fonction, $t@t$ une application, $null$ dénote l'ensemble vide, et $t \wr t$ une séquence. Dans le Rho-calcul, l'application est une opération décrite au même niveau du calcul que l'abstraction, c'est-à-dire les règles de réécriture. Nous obtenons ainsi un calcul similaire au Lambda-calcul avec motifs de S. P. Jones [58] où l'abstraction est faite non seulement par rapport à une variable mais en considérant aussi le contexte de la variable. Ce type d'abstraction nous donne une information purement syntaxique sur le contexte de la variable et, dans le Rho-calcul, la théorie de filtrage décrit le comportement des symboles de la signature. Ceci nous permet d'exprimer implicitement des propriétés pour les symboles comme, par exemple, l'associativité et la commutativité de l'opérateur "+" de l'arithmétique.

Le résultat d'une réduction dans le Rho-calcul est soit un ensemble vide représentant l'échec de l'application, soit un singleton représentant un résultat déterministe, soit un ensemble ayant plusieurs éléments représentant un choix non-déterministe de résultats. Par exemple, dans une théorie purement syntaxique, le ρ-terme $(X \to X)@a$ modélise le λ-term $(\lambda X.X)\,a$ et se réduit vers a ; le ρ-terme $(f(X\ Y) \to X)@f(a\ b)$ se réduit vers a ; le ρ-terme $(f(X\ Y) \to X)@g(c)$ se réduit vers l'ensemble vide *null* (échec) ; le ρ-terme $(f(X\ Y) \to X \wr f(X\ Y) \to Y)@f(a\ b)$ se réduit vers l'ensemble $a \wr b$.

– Dans [27] nous avons proposé le Rho-calcul dans un cadre de typage à la Church, en présentant huit systèmes des types placés dans un *Rho Cube* à la Barendregt. Le plus puissant de ces systèmes s'inspire essentiellement du *Extended Calculus of Constructions* (ECC) de Z. Luo [53].



On pense que l'étude du Rho-calcul typé ou non est fondamentale et prometteuse afin d'explorer différentes sémantiques des langages à objets, et d'étendre le Calcul de Constructions (et peut-être aussi Coq) avec une solide notion de réécriture (voir aussi [8]).

– **Formalisation des objets à travers la réécriture** Le Rho-calcul peut être employé pour donner une sémantique à l'application des règles du langage Elan [18], un environnement de programmation et de prototypage reposant sur la logique de réécriture associée à la notion de stratégie. Dans [26, 29], avec Claude Kirchner, Horatiu Cirstea, et Benjamin Wack nous avons montré comment le Rho-calcul peut être utilisé en tant que *lingua franca* pour capturer différents paradigme de programmation, comme par exemple la programmation à objets ; en fait, l'interprétation dénotationnelle de l'appel d'une méthode m sur un objet *obj* dans la programmation à objet (Kamin [43]) via `self` (`this` de Java) peut être simplement représentée en Rho-calcul en terme de filtrage et application. Par exemple, si on utilise l'abréviation suivante pour modéliser l'auto-application (*self-application* des langages à objets) :

$$t_1.t_2 \stackrel{\text{def}}{=} t_1@t_2@t_1$$

alors un simple objet *Point* avec un champ *val* et deux méthodes *get* et *set* sera représenté en Rho-calcul avec :

$$\begin{aligned} Point \stackrel{\text{def}}{=}\ & val{\to}S{\to}v(1\ 1) \wr \\ & get{\to}S{\to}S.val \wr \\ & set{\to}S{\to}v(X\ Y){\to}(S \wr val{\to}S'{\to}v(X\ Y)) \end{aligned}$$

et un simple appel de méthode sera représenté par le calcul suivant :

$$\begin{aligned} Point.get \stackrel{\text{def}}{=} & Point@get@point \mapsto\!\!\!\!\to \\ & (get \to S \to S.val)@get@Point \to (S \to S.val)@Point \to \\ & Point.val \stackrel{\text{def}}{=} Point@val@Point \mapsto\!\!\!\!\to (val \to S \to v(1\ 1))@val@Point \to \\ & (S \to v(1\ 1))@Point \to v(1\ 1) \end{aligned}$$

et une simple classe *PointClass* sera représenté en Rho-calcul avec :

$$\begin{aligned} PClass \stackrel{\text{def}}{=} & \ new \to S \to (val \to S' \to (S.preval)@S' \wr \\ & \qquad\qquad get \to S' \to (S.preget)@S' \wr \\ & \qquad\qquad set \to S' \to (S.preset)@S') \wr \\ & \ preval \to S \to S' \to v(1\ 1) \wr \\ & \ preget \to S \to S' \to S'.val \wr \\ & \ preset \to S \to S' \to v(X\ Y) \to (S'.val := S'' \to v(X\ Y)) \end{aligned}$$

où

$$\begin{aligned} obj.m \stackrel{\text{def}}{=} & \ obj@m@obj \quad \text{Kamin } \textit{self-application} \\ (a.m := b) \stackrel{\text{def}}{=} & \ (m \to b \wr a) \ \textit{update} \text{ fonctionnel} \\ (a.m := b) \stackrel{\text{def}}{=} & \ (m \to b \wr kill_m@a) \ \textit{update} \text{ impératif grâce à } kill_m \\ kill_m \stackrel{\text{def}}{=} & \ (X \wr m \to Z \wr Y) \to X \wr Y \end{aligned}$$

et

$$PClass.new \mapsto\!\!\!\!\to Point$$

Le Rho-calcul est également un excellent candidat pour intégrer le paradigme à objets dans un calcul de réécriture ; en fait, l'interprétation dénotationnelle de l'appel d'une méthode m sur un objet *obj* dans la programmation à objet (Kamin [43]) peut être simplement représentée en Rho-calcul en terme de filtrage et application. La figure **??** présente, par exemple, le codage en Rho-calcul d'un simple objet *Point* et d'une simple classe *PointClass* [6] (on présente soit un codage fonctionnel, soit un codage impératif, grâce au ρ-term $kill_m$, qui sélectionne une méthode dans l'objet et l'élimine).

– Écriture d'un livre sur le calcul de réécriture [64] avec les membres du projet Protheo.

1.5 Les substitutions explicites

Un autre veine de recherche concerne le rapport entre la modélisation des langages de programmation, l'étude de la substitution en lambda-calcul, et en particulier les substitutions explicites. Les substitutions explicites, introduites par Abadi, Cardelli, Curien et Lévy, permettent une description du lambda-calcul où substitution et élimination de beta-redex sont décrites par les règles d'un même système de réécriture. Les calculs ainsi obtenus permettent donc de décrire des stratégies de substitution, lesquelles interagissent naturellement avec les stratégies du lambda-calcul.

L'avantage principal des calculs de substitution explicite est de permettre la modélisation et l'implantation d'outils dans lesquels la substitution n'est pas systématiquement appliquée

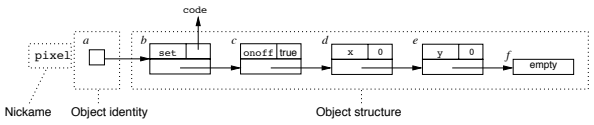
aux variables, mais bloquée dans le terme, soit parce que sa propagation serait inefficace (c'est le cas des implantations de langages fonctionnels), soit parce qu'elle est impossible.

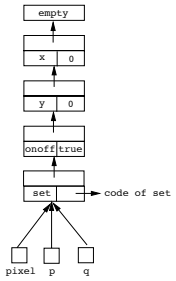
Les substitutions explicites peuvent être utilisées aussi pour la définition d'un système général pour la fondation d'une sémantique opérationnelle des calculs à objets, en insistant sur des problèmes propres aux calculs fonctionnels ou impératifs. Donc cette recherche est strictement corrélée avec le sujet de recherche présenté à la Sous-section **??**.

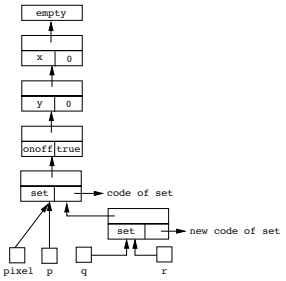
Les substitutions explicites sont beaucoup utilisées dans le contexte de la démonstration automatique [41]. Par exemple, une preuve incomplète est un lambda-terme dont certains composants ne sont pas encore connus. De tels composants sont modélisés par des variables instanciables par des termes quelconques, nommées méta-variables. La substitution ne peut généralement pas être définie sur un tel type de variables car on ne connaît pas encore la structure du terme qui va l'instancier. Les substitutions explicites sont donc une solution pour la normalisation des preuves incomplètes. Il est par conséquent important de disposer d'un calcul avec substitution explicite qui soit confluent sur les méta-termes (ou termes avec méta-variables) et qui *préserve* la forte normalisation du lambda-calcul.

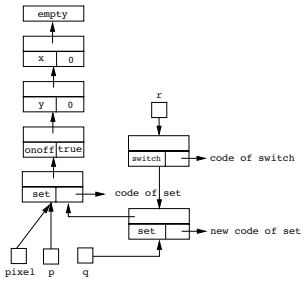
1.5.1 Contribution majeures

– **Formalization des langages à objets à travers la réécriture de graphes**. Dans [32] avec Pierre Lescanne (Ens Lyon) et Daniel Dougherty (Welsleyan, USA), on a présenté un formalisme appelé *Addressed Term Rewriting Systems (ATRS)* qui peut être employé pour l'implementation de langage de programmation et assistant à la preuve qui possédent ou manipulent de structures de données cycliques avec *partage*. Les ATRS sont bien adapté pour décrire des calculs ou langages à objets. Dans ce papier on présente un langage à objets avec des traits fonctionnels, appelé $\lambda\mathcal{O}bj^a$. Une sémantique operationnelle avec substitution explicite et un systeme de type statique qui assure l'absence de l'erreur run-time `does-not-understood` sont présenté.









1.6 Analyse du contrôle

Au cours de mon stage de DEA [46], j'ai étudié la capacité d'un programme à accéder à son propre flux d'évaluation. Cette possibilité est importante si nous voulons maintenir l'efficacité, la lisibilité et améliorer l'expressivité d'un logiciel. Les *opérateurs de contrôle* permettent de modifier l'ordre normal d'évaluation d'un logiciel. Un grande nombre d'opérateurs de contrôle et *exception handlers* peuvent être trouvés dans les langages fonctionnels, impératifs, et logiques [14] (par exemple `call-cc, raise / exception, catch / throw, on-exception, raise-exception`). Cette exploration a comme partie formelle l'étude des *continuations* et des sémantiques dénotationnelles avec les continuations. Les continuations sont aussi utilisées pour modéliser les sémantiques dénotationnelles avec exceptions dans le cadre des langages logiques [52].

L'étude de sémantiques avec les continuations est enfin importante dans la construction de compilateurs ; un langage de haut niveau est très souvent traduit dans un langage intermédiaire à l'aide de transformations en *continuation passing style* (CPS).

1.6.1 Contribution majeures

– **Opérateurs de controle dans ProLog.** Les langages logiques *à la* Prolog offrent des construction de langage très simples pour manipuler le contrôle d'un programme logique (par exemple le *cut*). Le contrôle d'un programme logique peut aisément être modélisée grâce aux semantique dénotationelle avec le *continuations*. Avec M. L. Sapino (Un. Turin) nous avons conçu une sémantique dénotationnelle avec continuations pour modéliser un petit langage logique avec les operateurs de contrôle `onexception,raisexc`, qui permettent au programmeur de modifier explicitement l'ordre d'évaluation des clauses logiques. Ces opérateurs ont quelque similitude avec ceux qui sont définis dans le *SICStus* Prolog. Par exemple dans le simple programme logique

```
  A:- onexception(X,B_1,H),C.
B_1:- B_2, B_3.
B_3:- (equal? n), raise_exc(X); B_1.
B_2:-...
  H:-...
  C:-...
```

La sémantique de ce simple programme est la suivante : lorsque le but `A` est évalué, `onexception(x,B_1,H)` est appelé : l'étiquette `X` d'exception est mémorisée, et le but `B_1` est appelé dans l'environnement résultant ; intuitivement, le but `onexception(x,B_1,H)` a succès si $\mathtt{B_1}$ à succès, sinon une exception est soulevé et le but `H` est évalué. L'évaluation de `B_1` engendre l'évaluation de `B_2` (non indiqué) qui termine avec succès. Le but `B_3` est ensuite évalué ; puisque il est une disjonction, son premier disjoint est exploré. Si le test (`equal?n`) échoue, alors le premier disjoint de `B_3` échoue, et le but `B_1` est évalué à nouveau. Si le test (`equal?n`) réussit alors une exception est levée via le but `raise_exc(X)`. L'interprète accède à la pile d'exécution à la recherche d'une exception declarée et nommée `X`. Donc le traitement d'exception décrit en `H` peut être évalué. Si `H` termine avec succès , alors le contrôle passera au but `C`, sinon l'échec de `H` fera suivre l'échec du but global `A`. Comme résultat auxiliaire à l'introduction des opérateur de contrôle, on a montré comment le *cut* peut être simulé facilement à travers ces deux opérateurs.

– Dans l'interprète du calcul de réécriture Snake que j'ai écrit en 2005 [50], un mecha-nisme d'exception est présenté. Le méchanisme est constitué de deux regles de réecriture simples avec deux opérateurs de "fleche" différent (`:-)` et `(- :)` :

`Handler :-) Protected` et `(- : raise-value`

Intuitivement, on évalue le système de réécriture `Protected` ; pendant cet évaluation, l'évaluation du terme algébrique `( :- raise-value` comportera la sortie immédiate du système de réécriture `Protected`, ainsi que l'évaluation du système de récriture `Handler`. Comme example (dans la syntaxe de Snake) on présente un programme qui prend une liste d'entiers et produit la liste inversée (si pas de zero) ou le prémier zero rencontré :

```
[HAND = Y -> Y |
  REV = (   (nil,R) -> R,
        ((X\0,T),R) -> (REV (T,(X,R))),
        ((0,T),R)   -> (-: 0)];;
```

Pour le deux listes

```
LIST1 = (1,2,3,4,5,7,8,9,10,nil)
LIST2 = (1,2,3,0,5,7,8,9,10,nil)
```

l'évaluation de `HAND :-) (REV (LIST1,(nil)))` produit comme résultat le liste inversé et l'évaluation de `HAND :-) (REV (LIST2,(nil)))` produit simplement `0`.

– Dans le papier *Rewriting Calculi with(out) types* [28], j'ai présenté une sémantique naturelle *à la* Kahn pour exprimer un mécanisme d'exception suite à un échec de filtrage. La syntaxe du mécanisme d'exception est la suivante :

`try Protected catch [term1 match term2] with Handler`

Intuitivement `term1` et `term2` sont deux termes algébriques qui ne filtrent pas. Si, lorsque l'évaluation du système de réécriture `Protected` un échec de filtrage définitif `term1` $\ll$ `term2` est rencontré, alors l'exécution de `Protected` est arrêté et le système de réécriture `Handler` sera évalué, sinon l'évaluation de `Protected` continuera. L'ensemble de valeurs de "sortie" $\mathcal{O}$ sera modifié en rajoutant le valeur $[\mathcal{T}_1 \ll \mathcal{T}_2]$ qui dénote un échec de filtrage.

Les règles les plus intéressantes dans la sémantique big-step *à la* Kahn sont les suivantes :

$$\frac{\nexists \sigma.\ \sigma(\mathcal{T}_1) \equiv \mathcal{T}_2}{(\mathcal{T}_1 \to \mathcal{T}_3)@\mathcal{T}_2 \Downarrow [\mathcal{T}_1 \langle\!\langle \mathcal{T}_2]} (\text{Red}-\sigma_2)$$

$$\frac{\mathcal{T}_1 \Downarrow [\mathcal{T}_2 \langle\!\langle \mathcal{T}_3] \quad \mathcal{T}_4 \Downarrow \mathcal{O}}{\mathsf{try}\ \mathcal{T}_1\ \mathsf{catch}\ [\mathcal{T}_2 \langle\!\langle \mathcal{T}_3]\ \mathsf{with}\ \mathcal{T}_4 \Downarrow \mathcal{O}} (\text{Red}-\text{Exc}_1)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathcal{O} \quad \mathcal{O} \not\equiv [\mathcal{T}_2 \langle\!\langle \mathcal{T}_3]}{\mathsf{try}\ \mathcal{T}_1\ \mathsf{catch}\ [\mathcal{T}_2 \langle\!\langle \mathcal{T}_3]\ \mathsf{with}\ \mathcal{T}_4 \Downarrow \mathcal{O}} (\text{Red}-\text{Exc}_2)$$

Intuitivement :
- la règle ($Red{-}\sigma_2$) dit que si le deux termes ne filtrent pas, alors un exception est levée ;
- la règle ($Red{-}Exc_1$) *capture* l'exception et déclenche le traitement d'exception $\mathcal{T}_4$, si l'échec de filtrage a été déclaré dans la portée du `try-catch` ;
- la règle ($Red{-}Exc_2$) propage l'exception vers un `try-catch` déclaré plus à l'extérieur, si l'échec de filtrage n'a pas été déclaré dans la portée du `try-catch`.

1.7 Un réseaux de recouvrement

Cette dernière année j'ai proposé l'architecture du système de communication Arigatoni, qui permet d'implémenter dans l'Internet le paradigme du *Global Computing*. La communication entre les différentes unités du modèle s'effectue par le biais d'un protocole de communication simple, qui opère directement au dessus des protocoles TCP ou UDP. Les ordinateurs globaux de base communiquent en s'enregistrant tout d'abord à un service de courtage, ils peuvent ensuite proposer ou demander des services de manière interchangeable, selon une stratégie similaire au modèle de coopération *tit-for-tat* de Rapoport. Dans notre modèle, les ressources sont encapsulées dans l'intranet dans lequel elles résident, et les demandes pour des ressources qui se trouvent dans d'autres "intranets" sont effectuées au moyen d'une négociation courtier-à-courtier, en utilisant les mécanismes classiques de PKI. Ce modèle permet de réaliser divers scénarios d'informatique globale, depuis les applications classiques pair-à-pair telles que le partage de fichier, ou encore le partage de bande, jusqu'à des applications plus sophistiquées d'informatique en grille, telles que les gros (ou petits) calculs distribués à distance. Nous pouvons même envisager des scénarios réels de migration de calculs, dans l'esprit du langage de programmation Obliq, développé par Luca Cardelli.

1.7.1 Contribution majeures

Arigatoni [11, 30, 24, 25] est un modèle de communication léger pour la découverte de ressource dynamique. Inspiré du paradigme Publier/Souscrire, le modèle Arigatoni implémente un réseau recouvrant pour la découverte de ressources. Les entités dans Arigatoni sont organisées dans des *Colonies*. Une colonie est une organisation virtuelle simple composée d'exactement un leader, qui offre des services de type courtier, et un ensemble d'*Individus*. Les Individus sont des sous-colonies d'Individus, ou des unités basiques appelées des *Ordinateurs Globaux*. Les ordinateurs globaux communiquent en s'enregistrant tout d'abord à la Colonie,

ils peuvent ensuite demander et offrir des services de manière interchangeable. Le leader, appelé *Routeur Global*, doit analyser les requêtes ou les réponses arrivant de sa propre Colonie ou d'une Colonie voisine, et router les requêtes ou réponses vers d'autres Individus. Une fois cette phase de découverte achevée, les Individus entrent en contact les uns avec les autres sans d'avantage d'intervention de la part du système, suivant le modèle pair-à-pair. Les communications entre les unités actives du modèle s'effectuent au moyen d'un simple protocole. Arigatoni offre une découverte de ressources entièrement décentralisée, aynchrone et extensible, et peut être utilisé à des fins diverses, depuis les applications pair-à-pair jusqu'aux applications plus sophistiquées utilisées dans les grilles. Le principal objectif de cet article est de présenter le mécanisme de découverte de ressources utilisé dans le modèle Arigatoni, accompagné de simulations qui montrent que la découverte de ressources dans Arigatoni est efficace et extensible. Pour résumer :

- Organisation virtuel de *peers*
- Déecouverte des ressources entre *peers*
- Organisation et évolution *transparent*
- Difficulté : mis à jour des tables de routages
- organisation en colonies avec un leader et des peers
- découverte de ressources dans un *multi-layer* overlay

1.8 Sur la suite du document

Dans la suite du document je vous présenterai *Peter, le langage à objets qui n'existe pas* .... Peter contient des aspects innovants concernant la programmation typée à objets que j'ai étudiée ces dernières années. La démarche que je suivrai consiste à essayer de limiter les détails concernant les aspects théoriques de Peter (ensemble complet des règles de typage, suites de théorèmes abscons, etc.). En d'autre mots, je pense inutile de présenter Peter en me contentant d'une traduction brutale des différents articles que j'ai publiés.

La description de Peter sera fondée sur le trois "dogmes" suivants :

(Incrémentalité) Je commencerai par le plus petit fragment *complet au sens de Turing* et orienté vers les objets que j'appellerai *Baby Peter* ; je continuerai de façon *modulaire* d'extension en extension (c.à.d. chaque extension de Peter sera indépendante de la précédente). Baby Peter est essentiellement un sous-ensemble de Java qui est, *a une exception prés*, égal à *Featherweight Java* de Igarashi-Pierce-Wadler ; la différence principale est un mécanisme d'exceptions *ad hoc*, à ma connaissance jamais publié dans aucun article scientifique, qui permettra simplement au programmeur de décider quel système de type utiliser (voir point [*Type-programmable*]). Les chapitres présenteront les différents extensions de Baby Peter. Dans certains chapitres l'extension se focalisera seulement sur un nouveau système de type, tandis que dans d'autres chapitres l'extension sera associée à une extension de la syntaxe ET du système de types. Parmi les extensions je présenterai une extension de Peter proposant une forme nouvelle d'héritage multiple, une extension de Peter qui permettra à un objet de "s'échapper de sa classe" (c.à.d. de changer de rôle, en lui ajoutant dynamiquement des nouvelles méthodes), une extension de Peter avec filtrage évolué et enfin une extension de Peter qui permettra de mélanger *algorithmes et preuves de correction d'algorithmes*.

(`verbatim-like`) Plutôt que d'asséner à mes lecteurs une traduction française mot-à-mot de mes articles scientifiques, qu'il trouvera cependant en annexe en version originale, j'ai

opté pour une présentation de la syntaxe de chaque extension de Peter en style prolixe, tandis que la règle clé du système de typage de l'extension (il y en a toujours une ...) sera présentée de façon très simple, en ajoutant immédiatement des exemples pour motiver et comprendre son utilisation correcte.

(***Type-programmable***) Les systèmes de types des langages de programmation sont fixés par leurs concepteurs et ne sont pas des *objets* de "première classe" qui peuvent être modifiés ou simplement utilisés par le programmeur qui en subit les qualités et les faiblesses. À ma connaissance, aucun langage permet au programmeur de "programmer" sa discipline de type personnelle (à ses risques et périls bien sûr).

L'idée de modifier la discipline de typage à la compilation n'est pas nouvelle ; elle a été proposée par Gilad Bracha (Sun Labs) dès 2004, mais elle n'a jamais été formalisée. Intuitivement, il faudrait délimiter syntaxiquement un programme en se servant d'une construction syntaxique qui pourrait avoir la forme suivante :

```
usetypesystem{typesystem}{expression}
```

où `typesystem` est un système de type particulier (ni seul ni unique) qui sera disponible dans la distribution du langage (ou sera peut-être défini par le programmeur lui-même), tandis que `expression` sera une expression (un programme) qui devra être vérifiée par `typesystem`. Le programmeur pourra utiliser les différents systèmes de types que le langage Peter possédera dans ses librairies. La portée de cette construction syntaxique devra être *lexicale* avec des règles de *tolérance* entre différents systèmes de type.

Par exemple, considérons deux système de types, le premier appelé `normalising` et le deuxième appelé `recursive`. Une `expression` (sans entrée/sortie) qui sera typable avec `normalizing` terminera sûrement à l'exécution, tandis qu'une expression qui sera typable avec `recursive` pourra implémenter la récursion et, donc, a priori ne pas terminer. La relation de tolérance pourrait affirmer que si une expression est compilée avec le système `normalizing`, alors elle pourrait être utilisée dans une autre expression qui sera compilée avec le système `recursive` (l'affirmation réciproque n'étant pas vraie). Par exemple :

```
usetypesystem{recursive}{expression_1
                         ...
                         usetypesystem{normalizing}{expression_k}
                         ...
                         expression_n}
```

compilera si `expression_1,2,...k-1,k+1,...n` compilent dans `recursive` et `expression_k` compile dans `normalize`. En fait, on peut tolérer la routine `expression_k` (qui termine) dans un programme récursive entier qui potentiellement pourrait ne jamais terminer. Par contre, la relation de tolérance ne pourra pas accepter la réciproque, c.à.d. que le programme suivant ne sera pas typable ;

```
usetypesystem{normalizing}{expression_1
                           ...
                           usetypesystem{recursive}{expression_k}
                           ...
                           expression_n}
```

car on ne pourra jamais accepter (à moins d'enrichir Peter par des conditions pre-post contrôlées statiquement, ce qui est un problème difficile) une routine récursive dans un programme qui terminera grâce à une discipline de typage très rigide.

La possibilité de permettre au programmeur d'écrire sa propre discipline de typage et de l'utiliser à la volée est par elle-même un sujet intéressant qu'à ma connaissance, personne n'a jamais publié ; ceci constituera une autre contribution originale dans ma thèse d'habilitation.

Bibliographie

[1] M. Abadi. Baby Modula–3 and a Theory of Objects. *Journal of Functional Programming*, 4(2) :249–283, 1994.

[2] M. Abadi and L. Cardelli. A Theory of Primitive Objects : second-order systems. In *Proc. of ESOP*, LNCS. Springer Verlag, 1994.

[3] M. Abadi and L. Cardelli. A Theory of Primitive Objects : untyped and first order systems. In *Proc. of TACS*, volume 789 of *LNCS*, pages 296–320. Springer Verlag, 1994.

[4] M. Abadi and L. Cardelli. An Imperative Object Calculus. In *Proc. of TAPSOFT/FASE*, LNCS, pages 471–485. Springer Verlag, 1995. Also in *Theory and Practice of Object Systems* 1(3) :151-166, 1995.

[5] M. Abadi and L. Cardelli. On Subtyping and Matching. In *Proc. of ECOOP*, volume 952 of *LNCS*, pages 145–167. Springer Verlag, 1995.

[6] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[7] M. Abadi and L. Cardelli. A Theory of Primitive Objects : Untyped and First Order Systems. *Information and Computation*, 125(2) :78–102, 1996.

[8] F. Barbanera, M. Fernandez, and H. Geuvers. Modularity of Strong Normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 7(6) :613–660, 1997.

[9] H. Barendregt. Lambda Calculi with Types. In S. Abramsky, Dov.M. Gabbai, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 118–310. Oxford University Press, 1992.

[10] B Barras. Coq in coq. Rapport Technique 3026, INRIA, Roquencour, 1996.

[11] D. Benza, M. Cosnard, L. Liquori, and M. Vesin. Arigatoni : A Simple Programmable Overlay Network. In *Proc. of John Vincent Atanasoff International Symposium on Modern Computing*, pages 82–91. IEEE, 2006.

[12] G. Blashek. Type-safe oop with prototypes : the concepts of Omega. *Structured Programming*, 12(12) :1–9, 1991.

[13] V. Bono and M. Bugliesi. Matching Constraints for the Lambda Calculus of Objects. In *Proc. of TLCA*, volume 1210 of *LNCS*, pages 46–62. Springer Verlag, 1997.

[14] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping for Extensible, Incomplete Objects. *Fundamenta Informaticae*, 38(4) :325–364, 1999.

[15] V. Bono, M. Bugliesi, and L. Liquori. A Lambda Calculus of Incomplete Objects. In *Proc. of MFCS*, volume 1113 of *LNCS*, pages 218–229. Springer Verlag, 1996.

[16] V. Bono and K. Fisher. An Imperative, First-Order Calculus with Object Extension. In *Proc. of ECOOP*, number 1445 in LNCS, pages 462–497. Springer Verlag, 1998.

[17] V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proc. of CSL*, volume 933 of *LNCS*, pages 16–30. Springer Verlag, 1995.

[18] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of elan. In *Proc. of WRLA*, volume 15. Electronic Notes in Theoretical Computer Science, 1998.

[19] K. Bruce, L. Cardelli, and C. B. Pierce. Comparing Object Encoding. In *Proc. of TACS*, volume 1281 of *LNCS*, pages 415–438. Springer Verlag, 1997.

[20] M. Bugliesi, G. Delzanno, L. Liquori, and M. Martelli. A Linear Logic Calculus of Objects. In The MIT Press, editor, *Proc. of JICSLP*, pages 67–81, 1996.

[21] M. Bugliesi, G. Delzanno, L. Liquori, and M. Martelli. Object Calculi in Linear Logic. *Journal of Logic and Computation*, 10(1) :75–104, 2000.

[22] L. Cardelli. Obliq : A Language with Distributed Scope. *Computing Systems*, 8(1) :27–59, 1995.

[23] G. Chambers. The Cecil language specifications and rationale. Technical Report 93-03-05, University of Washington, Dept. of Computer Science and Engineering, 1993.

[24] R. Chand, M. Cosnard, and L. Liquori. Resource Discovery in the Arigatoni Model. Technical Report 5924, INRIA, 2006.

[25] R. Chand, M. Cosnard, and L. Liquori. Resource Discovery in the Arigatoni Overlay Network. In *I2CS : International Workshop on Innovative Internet Community Systems*, volume LNCS. Springer, 2006. To appear. Also available as RR INRIA 5928.

[26] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In *Proc. of RTA*, volume 2030 of *LNCS*, pages 168–183. Springer Verlag, 2001.

[27] H. Cirstea, C. Kirchner, and L. Liquori. The rho Cube. In *Proc. of ESOP/FOSSACS*, volume 2051 of *LNCS*, pages 77–92. Springer Verlag, 2001.

[28] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting Calculus with(out) Types. In *Proc. of WRLA*, ENTCS, 2002.

[29] H. Cirstea, L. Liquori, and B. Wack. Rewriting calculus with fixpoints : Untyped and first-order systems. In *Proc. of TYPES*, volume 3085. Springer, 2003.

[30] M. Cosnard, L. Liquori, and R. Chand. Virtual Organizations in Arigatoni. *DCM : International Workshop on Developpment in Computational Models. Electr. Notes Theor. Comput. Sci.*, 2006. To appear.

[31] D. Dougherty, F. Lang, P. Lescanne, L. Liquori, and K. Rose. A generic object-calculus based on addressed term rewriting systems. In *Proc. of Westapp*, pages 6–25. Logic Group Preprint series No 210. Utrecht University, the Netherlands, 2001.

[32] D. J. Dougherty, P. Lescanne, and L. Liquori. Addressed term rewriting systems : Application to a typed object calculus. *Journal of Mathematical Structures in Computer Science*, 16(4) :667–709, 2006.

[33] K. Fisher. *Type System for Object-Oriented Programming Languages*. PhD thesis, University of Stanford, 1996. `ftp ://theory.stanford.edu/pub/kfisher/thesis.ps.gz`.

[34] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1) :3–37, 1994.

[35] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT*, volume 965 of *LNCS*, pages 42–61. Springer Verlag, 1995.

[36] K. Fisher and J. H. Reppy. The design of a class mechanism for moby. In *Proc. of PLDI*, volume 34 of *SIGPLAN Notices*, pages 37–49. The ACM Press, 1999.

[37] K. Fisher and J. H. Reppy. A calculus for compiling and linking classes. In *Proc. of ESOP*, volume 1782 of *LNCS*, pages 153–149. Springer Verlag, 2000.

[38] K. Fisher and J. H. Reppy. Extending moby with inheritance-based subtyping. In *Proc. of ECOOP*, volume 1850 of *LNCS*, pages 83–107. Springer Verlag, 2000.

[39] P. Di Gianantonio, F. Honsell, and L. Liquori. A Lambda Calculus of Objects with Self-inflicted Extension. In *Proc. of OOPSLA*, pages 166–178. The ACM Press, 1998.

[40] W. A. Howard. The Formulae-as-Types Notion of Construction. In J. P. Seldin and J. R. Hyndley, editors, *To H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[41] G. Huet, G. Kahn, and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial, version 6.1. rapport technique 204, INRIA, 1997. Version révisée distribuée avec Coq. `http://coq.inria.fr/`.

[42] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java : A minimal core calculus for Java and GJ. In *Proc. of OOPSLA*, pages 132–146. The ACM Press, 1999.

[43] S. N. Kamin. Inheritance in smalltalk-80 : A denotational definition. In The ACM Press, editor, *Proc. of POPL*, pages 80–87, 1988.

[44] F. Lang, P. Lescanne, and L. Liquori. A framework for defining object-calculi (extended abstract). In *Proc. of FM*, volume 1709 of *LNCS*, pages 963–982. Springer Verlag, 1999.

[45] D. Leivant. Polymorphic Type Inference. In *Proc. of POPL*, pages 88–98. The ACM Press, 1983.

[46] L Liquori. Semantica e Pragmatica di un Linguaggio Funzionale con le Continuazioni Esplicite. Laurea in science dell'informazione, Université de Udine, Italie, 1990.

[47] L. Liquori. An Extended Theory of Primitive Objects. Technical Report CS-23-96, Computer Science Department, University of Turin, Italy, 1996.

[48] L. Liquori. An Extended Theory of Primitive Objects : First Order System. In *Proc. of ECOOP*, volume 1241 of *LNCS*, pages 146–169. Springer Verlag, 1997.

[49] L. Liquori. On Object Extension. In *Proc. of ECOOP*, volume 1445 of *LNCS*, pages 498–552. Springer Verlag, 1998.

[50] L. Liquori. Snake : The First Ascii-Oriented Script Language Based on the Original Screenplay of the Imperative Rewriting Calculus V1.1, `http://www-sop.inria.fr/mascotte/Luigi.Liquori/Snake`.

[51] L. Liquori and G. Castagna. A Typed Lambda Calculus of Objects. In *Proc. of Asian*, volume 1179 of *LNCS*, pages 129–141. Springer Verlag, 1996.

[52] L. Liquori and M.L. Sapino. Dealing with Explicit Exceptions. In *Proc. of GULP-PRODE, Joint Conference on Declarative Programming*, pages 296–308, 1994.

[53] Z. Luo. ECC : An Extended Calculus of Constructions. In *Proc. of LICS*, pages 385–395. IEEE Computer Society, 1990.

[54] J. C. Michell. Toward a Typed Foundation for Method Specialization and Inheritance. In *Proc. of POPL*, pages 109–124. The ACM Press, 1990.

[55] J. Noble, A. Taivalsaari, and I. Moore, editors. *Prototype-Based Object-Oriented Programming : Concepts, Languages, and Applications*. Springer Verlag, 1999.

[56] J. Palsberg. Efficient Inference of Object Types. In *Proc. of LICS*, pages 186–195, 1993.

[57] J. Palsberg and T. Jim. Type Inference for Simple Object Types is NP-Complete. *Nordic Journal of Computing*, 4(3) :259–286, 1997.

[58] S. Peyton-Jones. *The implementation of functional programming languages*. Prentice Hall, Inc., 1987.

[59] K. R. Rajendra, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald : a general-purpose programming language. *Software Practice and Experience*, 21(1) :91–118, 1991.

[60] D. Rémy. Refined Subtyping and Row Variables for Record Types. Draft, 1995.

[61] D. Rémy. From Classes to Objects via Subtyping. In *Proc. of ESOP*, volume 1381 of *LNCS*, pages 200–220. Springer Verlag, 1998.

[62] J.G. Riecke and C. Stone. Privacy via Subsumption. In *Electronic proc. of FOOL-98*, 1998. Also in *Theory and Practice of Object Systems*.

[63] A. Taivalsaary. Kevo, a prototype-based object-oriented language based on concatenation and module operations. Technical Report LACIR 92-02, University of Victoria, 1983.

[64] The Rho-Team. The Rewriting Calculus, 2007. In preparation, `rho.loria.fr`.

[65] D. Ungar and R. B. Smith. Self : The power of simplicity. In *Proc. of OOPSLA*, pages 227–241. The ACM Press, 1987.

[66] S. van Bakel, L. Liquori, S. Ronchi della Rocca, and P. Urzyczyn. Comparing Cubes of Typed and Type Assignment System. *Annals of Pure and Applier Logics*, 86(3) :267–303, 1997.

[67] Web site de l'ambassade de France au Étas-Unis. Catching bugs would save big bucks. Site web de l'ambassade de France au Étas-Unis, juillet 2002. `http://www.info-france-usa.org/fr`.