Mémoire d'habilitation à diriger des recherches

en informatique

présenté à

l'Université de Paris 7

par

Didier Rémy

Des enregistrements aux objets

Soutenu le 7 septembre 1998 devant le jury composé de :

MM. **Gérard Berry**
Guy Cousineau
Jean-Pierre Jouannaud
Claude Kirchner
Jean-Jacques Lévy
Benjamin Pierce
Christian Queinnec

Table des matières

Introduction

L'intelligence pour s'exercer a besoin d'un support linguistique. Ne serait-ce que pour devenir un nez, il faut d'abord apprendre le langage des arômes. De cette primordialité du langage dans la pensée, il n'est pas surprenant que les langages de programmation soient aussi un point d'articulation entre la conception d'un logiciel et sa réalisation. Bien que la structure globale, les algorithmes particuliers mis en oeuvre et le soin apporté au codage soient des facteurs essentiels, le langage de programmation est souvent lui-même déterminant pour la rapidité de la mise au point, la qualité et la clarté du code et la sûreté du logiciel.

Les langages de programmation dits généraux, c'est-à-dire non spécialisés à certains types de problèmes, sont complets au sens de Turing. Cela veut dire qu'ils permettent d'écrire tous les algorithmes calculables. Ils ne sont cependant pas tous équivalents, car ils diffèrent énormément par leur expressivité, c'est-à-dire leur capacité à exprimer succintement certains algorithmes. Nous sommes donc poussés à une recherche —sans fin— de nouvelles structures de programmation, plus abstraites, permettant de décrire des algorithmes de façon plus concise. Nous ne recherchons pas toutefois l'expressivité à tout prix et nous nous efforçons de n'introduire que des constructions générales, mais aussi simples et intuitives, modulaires, bien formalisées et sûres.

En contrepartie de la complétute des langages de programmation généraux, il faut abandonner tout espoir de pouvoir décider de propriétés importantes telles que la terminaison d'un programme, ou ce qui est équivalent, sa bonne exécution. Mais il ne faut pas pour autant abandonner la sûreté de l'exécution, sous prétexte de ne rien sacrifier à l'expressivité. Il est possible et important de conserver l'expressivité tout en se limitant à une sous-classe décidable des programmes qui s'évaluent normalement.

Après une étude approfondie du typage des enregistrements avec synthèse des types, nous proposons une extension du langage ML avec des objets et des classes, et les opérations les plus avancées qui leurs sont associées. Nous augmentons ainsi significativement l'expressivité du langage sans en perdre l'esprit.

Typage

Le typage est un outil général permettant de définir, par un critère simple et décidable, un sous-ensemble des expressions bien formées qui s'évaluent correctement. Il consiste à abstraire les détails d'un programme, les valeurs particulières prises par les arguments, et n'en retenir que leur structure. Par exemple, on distingue les entiers de type int des chaînes de caractères de type string, mais on confond tous les entiers entre eux et toutes les chaînes de caractères entre elles. Une fonction sur les entiers a le type int $\rightarrow$ int. Le lien précis entre le résultat et l'argument est alors perdu, les types des arguments et du résultat étant uniquement retenus. Ainsi, les fonctions successeurs ou prédécesseurs ont toutes deux le type int $\rightarrow$ int et, comme la preuve par neuf, le typage ne détectera pas les erreurs de signe.

Le *polymorphisme paramétrique* permet de donner à une expression un ensemble de types obtenus de façon uniforme. Ainsi, la fonction identité a pour type $\forall\alpha.(\alpha \to \alpha)$, ce qui signifie qu'elle a le type $\tau \to \tau$ pour tout type τ. Par exemple, elle a les types $\mathsf{int} \to \mathsf{int}$ et $\mathsf{string} \to \mathsf{string}$. Dans les langages *fonctionnels*, les fonctions peuvent être passées comme arguments d'autres fonctions ou retournées comme résultats. La fonction identité a donc aussi le type $(\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$. Le polymorphisme permet de retrouver une dépendance entre le type du résultat et le type de l'argument, mais moins précise bien sûr que le lien exact qui relie le résultat à l'argument.

Le polymorphisme non paramétrique est dit *ad hoc*. Par exemple, le polymorphisme de surcharge permet de regrouper sous un même nom plusieurs fonctions ayant des comportements différents. Le choix de la fonction effectivement appliquée est déterminé statiquement par le type précis de l'argument, voire pendant le calcul par la valeur de l'argument. Dans ce travail, nous nous intéresserons plus particulièrement au polymorphisme paramétrique et nous parlerons simplement de polymorphisme.

Le typage est en général *modulaire*. Par exemple, le type de l'application d'une expression à une autre peut être obtenu en typant séparément chacune des expressions puis en combinant les types obtenus. Plus généralement, il est souhaitable de pouvoir calculer le type d'un programme à partir des types des différentes parties qui le composent.

Une propriété intéressante mais difficile à satisfaire, donc qui n'est pas toujours vraie, est l'existence de *types minimaux* pour tous les programmes. Un type est minimal pour une expression lorsqu'il est correct et que tous les autres types corrects en sont des instances.

Enfin, un système de typage doit être conçu de façon à garantir la correction de l'évaluation. Cela signifie que l'évaluation des expressions peut se poursuivre jusqu'à l'obtention d'une valeur, ou indéfiniment, mais ne peut pas s'arrêter sur un calcul inachevé. En général, on montre la combinaison de deux propriétés simultanément plus fortes. D'une part le type d'un programme est préservé au cours du calcul (appelé réduction du programme). D'autre part les seuls programmes bien typés qui ne peuvent plus être réduits sont les valeurs.

Le système de types de ML possède toutes les propriétés énoncées ci-dessus. Il est simple, robuste et en fait relativement puissant : nous verrons dans les chapitres suivants comment l'étendre sans en perdre l'essence. Dans [102], le système de type de ML est enrichi en munissant l'algèbre des types d'une théorie équationnelle. Cette extension préserve l'harmonie et la simplicité de présentation du langage. De plus, on retrouve la version d'origine en considérant la théorie équationnelle vide. Le système de typage obtenu, plus riche, est à la base du typage des enregistrements et des objets, c'est-à-dire de l'essentiel des travaux présentés ici.

En fait, la simplicité de cette extension repose sur un enrichissement en largeur. Les théories équationnelles permettent d'augmenter la structure des types sans passer à l'ordre supérieur. L'ensemble des types possibles est ensuite restreint par des sortes. L'usage des sortes est important car en limitant les formes possibles des types, il simplifie remarquablement le problème de l'unification dans les théories équationnelles considérées. La combinaison de ces deux techniques s'avère très efficace. Elle est illustrée dans le chapitre 2 et appliquée au typage des enregistrements.

Synthèse des types et vérification de types

Dans le langage ML le typage est implicite, c'est-à-dire que les types ne font pas partie des programmes, mais sont synthétisés. C'est l'approche de Curry. D'autres langages sont, au contraire, explicitement typés. Les expressions n'ont plus de sens si les informations de types sont retirées ; le système de typage n'est alors utilisé que pour vérifier la cohérence des annotations de type : c'est l'approche de Church.

Cette différence n'est pas toujours essentielle. En effet, la sémantique d'un langage explicitement typé manipule les types, mais en général, les types ne participent pas activement à la réduction. Il est alors possible d'obtenir une présentation à la Curry par effacement des informations de types dans les expressions, puis en projetant la relation de réduction typée dans l'ensemble des termes non typés. Inversement, les expressions d'un langage à la Curry peuvent être enrichies par des informations de types, et possèdent donc une présentation naturelle à la Church.

D'ailleurs les visions de Curry et de Church sont deux positions extrêmes. Bien souvent, nous nous trouvons dans une situation intermédiaire où certaines informations de types sont explicites alors que d'autres restent implicites. La différence entre les deux approches n'est pas non plus absolument rigoureuse. Prenons comme exemple le langage ML, présenté comme un langage où les types sont entièrement synthétisés. Il permet des déclarations de types concrets qui consistent à introduire un nouveau constructeur de type et des fonctions de construction et de destruction pour les valeurs de ce type. L'utilisation d'une fonction de construction ainsi déclarée porte une information de type implicite si étroitement liée au constructeur que l'on pourrait considérer cette information comme explicite. Dire qu'une construction du langage est explicitement ou implicitement typée comporte donc une part de convention. Il serait sans doute plus approprié de comparer la quantité d'information de type portée, implicitement ou explicitement, par des constructions équivalentes dans des langages différents. Par exemple, les types concrets de ML sont typés plus explicitement que les variantes polymorphes dans lesquelles les constructeurs sont des étiquettes sans appartenance à un type particulier. De même, lorsqu'elles sont surchargées les opérations arithmétiques sont plus implicites que dans un langage où le symbole 1 est forcément un entier et le symbole + l'addition des entiers. En effet, cela reviendrait à lire 1 comme (1:int) dans un langage avec surcharge.

La synthèse des types dans le langage ML n'est donc pas totale. La différence entre la synthèse des types et leur vérification est aussi une différence de point de vue. Dire que ML est implicitement typé c'est dire que l'on suit l'approche de Curry, mais on ne peut pas pour autant définir les déclarations de types concrets sans introduire la notion de type. Inversement dire que le système F est explicitement typé, c'est reconnaître qu'une des constructions principales du langage, l'abstraction comporte obligatoirement une information de type.

L'augmentation de l'expressivité du système de typage, inéluctable, rend la synthèse des types plus difficile et nécessite plus d'annotations dans le programme source. Nous avons mentionné ci-dessus le cas des variantes polymorphes et celui de la surcharge. C'est évidemment le cas aussi pour l'ajout de polymorphisme d'ordre supérieur, pour lequel il n'est plus possible de synthétiser tous les types. Au mieux, nous ne pouvons plus que faire de la synthèse partielle des types. La difficulté est alors de permettre de ne pas indiquer certains types tout en conservant la propriété des types minimaux.

Dans ces travaux nous choisissons la présentation traditionnelle du langage ML en prenant le point de vue de Curry. Dans le chapitre 7 nous étendons ML avec des types d'ordre supérieur tout en préservant l'essence de ML, donc aussi l'approche à la Curry. Toutefois, de plus en plus de travaux s'orientent vers une approche typée. Sur le plan théorique, cela permet de présenter ML comme un sous-ensemble d'un langage explicitement typé plus puissant (le système F). Sur le plan pratique, cela permet de préserver les types pendant la compilation (donc pendant une certaine partie du calcul), même s'ils n'y participent pas activement.

Aussi, la frontière entre ces deux styles qui a été autrefois très nette ne cesse aujourd'hui de s'estomper. Plusieurs travaux récents vont dans le sens d'un rapprochement des deux points de vue, que ce soit en augmentant le langage ML avec des annotations de types explicites, ou en introduisant de la synthèse partielle des types dans le système F ou les langages qui en sont dérivés F^ω et $F^\omega_{<:}$ [96, 89, 37].

Les enregistrements *pas à pas*

La notion d'enregistrement est extrêmement simple et se retrouve dans la plupart des langages de programmation. Les enregistrements sont des produits à champs nommés. À la différence des tuples, ils permettent d'accéder aux composantes par nom plutôt que par position. Dans toute leur généralité, les enregistrements peuvent également être construits en ajoutant de nouveaux champs à des enregistrements déjà existants. Dans un langage non typé ils peuvent simplement être simulés par des listes d'associations clé-valeur. Les enregistrements sont donc une construction simple avec une sémantique simple. Leur typage monomorphe ne pose d'ailleurs aucune difficulté. Cependant, une étiquette ne peut plus appartenir qu'à un seul type d'enregistrement dans tout le programme.

L'intérêt, mais aussi les difficultés de typage, commencent avec les *enregistrements polymorphes*. Ceux-ci permettent de ne pas déclarer leur type préalablement à leur utilisation comme il faut le faire dans le langage ML. Puisque les étiquettes n'appartiennent plus à un enregistrement particulier, elles peuvent être utilisées librement pour construire des enregistrements, ou bien accéder à leurs composantes. On peut ainsi écrire une fonction d'*accès polymorphe*, c'est-à-dire une fonction qui projette sur une étiquette fixée tout enregistrement défini sur cette étiquette, indépendemment des autres étiquettes. On ne saurait parler d'enregistrements polymorphes sans cette construction primordiale. Une autre opération, plus difficile à typer, est l'*extension polymorphe*. Elle permet l'ajout d'un champ donné à un enregistrement quelconque, que ce champ soit ou non défini dans l'enregistrement initial. On parle alors d'*enregistrements extensibles*.

L'accès polymorphe est essentiel parce qu'il modélise l'envoi de messages dans la programmation à objets. L'opération d'extension, importante elle aussi, modélise l'héritage simple.

Les objets *un grand pas*

La notion de programmation avec objets est connue depuis longtemps. Elle a d'abord été motivée par le besoin d'écrire les programmes de manière plus modulaires, pour mieux les comprendre dans leur ensemble, les corriger plus facilement, ou les adapter, mais aussi pour pouvoir réutiliser certaines composantes dans des contextes différents.

La notion de module est apparue avec des motivations très similaires. Dans leur réalisation les modules sont assez éloignés des objets, bien qu'ils reposent sur des mécanismes analogues : les structures d'enregistrements, l'abstraction de type, le sous-typage.

D'apparence simple, les objets (au sens de la programmation à objets) sont réellement compliqués. Cela peut surprendre le novice qui ne connaît des objets que leur popularité, mais surprendra moins le lecteur averti plus conscient de la multitude et de la complexité des mécanismes mis en œuvre.

Un objet comporte avant tout des données, passives, présentées en général comme un enregistrement. Ce sont les *variables d'instance* dans le jargon des langages à objets. Celles-ci peuvent être accédées directement, comme les champs d'un enregistrement. Les données peuvent aussi être examinées indirectement, en invoquant une méthode de l'objet. Les méthodes peuvent être vues comme des fonctions opérant sur les données de l'objet, donc qui reçoivent l'objet lui-même en argument. Elles sont ajoutées à l'enregistrement des variables d'instance comme des champs supplémentaires. Cela permet aux méthodes d'appeler d'autres méthodes du même objet. Ainsi, l'envoi d'un message à un objet, (on dit aussi l'invocation d'une méthode d'un objet) consiste à sélectionner la fonction correspondante puis lui passer l'objet en argument. Cela induit une forme de récursivité qui introduit beaucoup de difficultés : le type d'un objet est celui d'une structure d'enregistrement dont certaines composantes, les méthodes, ont des types fonctionnels de domaine le type de l'objet

lui-même. C'est donc un type récursif.

D'autres opérations sur les objets viennent en compliquer davantage le concept. Par exemple, une forme d'héritage est en général réalisée par l'intermédiaire des classes. Les classes sont des modèles d'objets, c'est-à-dire des objets abstraits par rapport aux valeurs particulières des données (variables d'instance). Les objets sont alors créés par instanciation des classes. L'héritage est un mécanisme qui permet de construire de nouvelles classes à partir de classes déjà existantes. La combinaison de l'héritage et de la récursion nécessite un mécanisme supplémentaire, la *liaison tardive*, qui permet aux appels récursifs entre les méthodes d'être résolus à la création de l'objet plutôt qu'à leur définition. D'autres constructions moins fondamentales n'en sont pas moins difficiles, comme par exemple la possibilité de cacher des méthodes a posteriori, et ne sont pas encore entièrement résolus.

Des enregistrements aux objets *il n'y a qu'un pas*

Dans l'ensemble des travaux présentés ici, enregistrements et objets sont intimement liés. Dans un premier temps nous nous sommes attachés à mieux comprendre les objets. Pour cela nous en avons d'abord étudié une version dégradée : les objets enregistrements. Notre motivation était alors que les mécanismes fondamentaux de la programmation avec objets devaient correspondre aux opérations d'accès et d'extension dans les enregistrements. Cela a abouti sur des outils et des techniques robustes, nécessaires autant pour le typage des objets que pour celui des enregistrements.

Poursuivant dans cette direction, nous avons étudié les objets vus comme des enregistrements. Cette expérience largement positive, nous a permis d'exprimer toutes les constructions importantes des langages à objets. Au delà de notre attente, nous avons aussi pu mettre en œuvre facilement des mécanismes réputés difficiles comme l'héritage multiple. Cela n'a été possible qu'en s'appuyant fortement sur l'ensemble des outils mis au point pour les enregistrements. Il est aussi intéressant de remarquer qu'en retour, ce travail a augmenté notre compréhension des objets et notre habileté à les expliquer.

Une fois le terrain débroussaillé, munis des bons outils et de concepts simplifiés, nous avons alors pris la direction inverse, et proposé des opérations primitives sur les objets. Le langage Objective ML est une extension à objets puissante et parfaitement intégrée au langage ML, malgré toutes les difficultés imposées par la synthèse des types en ML. Cela a facilité l'élimination des points rugueux de la proposition précédente, notamment ceux reliés à l'affichage des types et le report d'erreurs, en utilisant un mécanisme d'abréviation automatique. Puis, nous avons à nouveau enrichi cette base solide avec des méthodes polymorphes pour donner aux classes paramétriques toute leur puissance.

Mais il restait une dernière étape importante : refaire le chemin inverse jusqu'au point de départ, les enregistrements, et se demander quel autre chemin aurait été possible. C'est cette sorte d'introspection fructueuse que nous proposons dans le chapitre 8. En nous libérant de la contrainte de la synthèse des types, mais en conservant le lien étroit entre objets et enregistrements nous proposons d'unifier le concept de classe à celui d'objet.

Ainsi, la thèse que nous défendons dans ce mémoire est que *les objets sont une forme enrichie des enregistrements*. Nous justifions alors *a posteriori* le terme objets-enregistrements puisque les enregistrements deviennent une forme dégénérée des objets. Notre thèse ne contredit pas l'idée plus traditionnelle que les objets sont des enregistrements de fonctions. Mais les objets ne sont pas *que* cela.

Le plus court chemin des objets aux enregistrements que nous présentons dans ce mémoire s'appuie sur un ensemble de travaux publiés dans des conférences ou des journaux. Chacun approfondit une ou plusieurs des notions décrites ci-dessus. Ces articles, présentés dans l'ordre chronologique le sont aussi dans l'ordre logique des dépendances. Chaque travail s'appuie en effet sur une partie en

général assez grande des travaux précédents et en prolonge certains aspects ou en explore des voies nouvelles.

Le langage ML est le témoin de cette unité thématique. L'ensemble des travaux est motivé par le souhait d'ajouter des objets à ML, qui sera réalisé dans le chapitre 6. Support de ces travaux, le langage ML en est donc aussi le premier bénéficiaire *in fine*. Mais chaque étape intermédiaire lui offre aussi des applications importantes : les objets enregistrements dans le chapitre 2, le typage de la concaténation des enregistrements dans le chapitre 4 qui systématise un style de programmation, ou encore le polymorphisme d'ordre supérieur dans le chapitre 7.

Cette étude va, bien sûr, au delà du langage ML. D'une part parce que les extensions proposées suggèrent et incitent à une meilleure intégration du langage ML avec un système de types d'ordre supérieur. D'autre part, la simplicité et l'expressivité du typage des objets en Objective ML renforce l'idée que l'utilisation des types-enregistrements et du polymorphisme des variables de rangée doit être privilégiée par rapport au sous-typage qui peut être ajouté ultérieurement. On peut donc espérer au-delà du langage ML une simplification des calculs d'objets primitifs avec typage explicite.

Notations

Comme certains chapitres ont été écrits en collaboration avec d'autres personnes, les notations et les conventions typographiques changent donc forcément d'un chapitre à un autre. Nous avons toutefois essayé de les uniformiser autant que possible. Il reste néanmoins des différences de notations ou de style incompatible d'un chapitre à l'autre, et les notations doivent toujours être comprises en prenant le chapitre comme unité.

Chapitre 1

Un raccourci

Ce chapitre est un survol, en langue française, du reste du mémoire, composé d'articles en langue anglaise. Toutefois, pour éviter un simple résumé des différents articles, nous avons choisi un style plus informel et un contenu technique plus léger permettant de décrire plus librement les intuitions qui sous-tendent les idées développeés. Nous nous permettons également de présenter certains chapitres principalement au travers d'exemples. À la différence des autres chapitres écrits dans l'ordre chronologique et insérés sans "retouches", nous profitons ici du recul qui nous est offert pour mettre l'accent sur les points les plus importants ou simplement moins connus. Nous espérons que ce mélange de raccourcis et de détours aidera le lecteur sans le perdre, et lui montrera ces travaux sous une lumière légèrement différente.

1.1 Le langage ML

Avant d'étudier plusieurs extensions sophistiquées de ML, nous nous devons de commencer par un bref rappel sur le langage ML lui-même. Sans apport technique, ce rappel permet d'introduire quelques notations, un peu de vocabulaire et le formalisme qui se retrouveront dans tous les chapitres suivants. Nous en profitons également pour mettre en avant les principes du langage sur lesquels s'appuient les différentes extensions.

Nous pouvons restreindre notre étude au noyau ML, composé du λ-calcul avec constantes et d'une construction de liaison.

$$a ::= z \mid \texttt{fun}\ (x)\ a \mid a\ a \mid \texttt{let}\ x = a\ \texttt{in}\ a \qquad\qquad z ::= x \mid c$$

Les identificateurs, désignés par la lettre z, regroupent les variables x et les constantes c. Cette décomposition permet de factoriser une partie de la présentation. Par de nombreux aspects, notamment pour tout ce qui concerne le typage, les constantes se comportent comme des variables.

D'autres constructions, notamment des constructions impératives, sont nécessaires dans un vrai langage de programmation. Afin qu'elles puissent être facilement ajoutées ultérieurement, nous choisissons une sémantique d'appel par valeur. La plupart des résultats qui seront présentés restent valides, parfois après adaptation, pour une stratégie de réduction arbitraire (par exemple, le chapitre 4 se place dans un cadre plus général). Cependant, pour l'étude du typage, la paramétrisation de la sémantique par une stratégie de réduction particulière apporte peu, tout en compliquant notablement la présentation.

Dans le λ-calcul sans constante les valeurs sont les fonctions. En présence de constantes, il faut leur ajouter les valeurs construites et les primitives partiellement appliquées. Pour les définir de

façon paramétrique, nous attribuons à chaque constante une arité fixe k et nous distinguons les constructeurs C des primitives f. Une constante d'arité k est une valeur si elle est appliquée au moins de k fois. Un constructeur d'arité k appliqué k fois est aussi une valeur. (Il serait possible d'utiliser la notion habituelle d'arité, plus rigide, qui impose exactement k applications, mais celle-ci compliquerait la formalisation en obligeant une forme d'application n-aire. Aussi nous préférons cette forme plus tolérante, qui par ailleurs est assez naturelle et revient à autoriser les applications partielles pour les constantes.)

$c^k ::= C^k \mid f^k$	Constantes d'ordre k.
$v ::= \texttt{fun}\ (x)\ a \mid C^k\ a_1\ \ldots a_k \mid c^k\ a_1\ \ldots a_q$	si $q < k$

Nous donnons une sémantique opérationnelle à petits pas pour ML. Elle est définie par un ensemble de contextes d'évaluation et des règles de réduction. Les contextes d'évaluation E sont définis par la grammaire suivante :

$$E ::= [] \mid E\ a \mid v\ E \qquad \text{Contextes d'évaluation}$$

Les radicaux sont de la forme (β_v) ou (δ). La β-réduction est définie par la règle :

$$(\texttt{fun}\ (x)\ a)\ v \xrightarrow{\beta_v} v[a/x]$$

qui est une version restreinte de la règle (β) à une stratégie d'appel par valeur. Les δ-réductions sont définies par des règles de la forme

$$f^k\ v_1 \ldots\ v_k \xrightarrow{\delta} a$$

et données en paramètre, définissant ainsi la sémantique des primitives. La relation de réduction $\longrightarrow$ est la fermeture des relations (β_v) et (δ) par la règle de congruence :

$$\frac{a_1 \longrightarrow a_2}{E[a_1] \longrightarrow E[a_2]}$$

On note $\xrightarrow{*}$ la fermeture transitive de $\longrightarrow$ (Par abus, on note aussi parfois simplement $\longrightarrow$ pour $\xrightarrow{*}$).

Les types sont les variables, les flèches et les types construits. Les types sont donc paramétrés par un ensemble de symboles de types donnés avec leur arité. Les schémas de types sont des types quantifiés par un ensemble de variables. Les environnements de types sont des fonctions partielles des variables dans les schémas de types.

$\tau ::= \alpha \mid \tau \to \tau \mid g(\bar{\tau})$	Types
$\sigma ::= \forall \bar{\alpha}.\tau$	Schémas de type
$A ::= \emptyset \mid A \oplus (z \mapsto \sigma)$	Environnements de typage

Un jugement de typage est de la forme $A \vdash a : \tau$ et signifie que dans le contexte A, le programme a a le type τ. Nous donnons une présentation du typage dirigée par la syntaxe dans la figure 1.1. Le système de typage est aussi paramétré par un environnement initial A_0 de domaine l'ensemble des constantes.

Il est possible de montrer la correction de la sémantique vis à vis du typage à condition toutefois que les sémantiques statiques et dynamiques des constantes soient cohérentes.

La correction du typage s'exprime par deux propriétés (à elles deux plus fortes que la correction) : auto-réduction et progression. La propriété d'auto-réduction signifie que les typages sont préservés par réduction. Définissons le typage d'une expression a comme l'ensemble des paires (A, τ) telles que $A \vdash a : \tau$, et notons $\subset$ le transfert naturel de la relation d'inclusion sur les ensembles de typages en une relation d'inclusion sur les expressions.

$$\text{(Var-Inst)}\ \frac{z : \forall\bar{\alpha}.\tau_0 \in A}{A \vdash z : \tau_0[\bar{\tau}/\bar{\alpha}]} \qquad \text{(Fun)}\ \frac{A \oplus (x \mapsto \tau_2) \vdash a : \tau_1}{A \vdash \texttt{fun}\ (x)\ a : \tau_2 \to \tau_1} \qquad \text{(App)}\ \frac{A \vdash a_1 : \tau_2 \to \tau_1 \qquad A \vdash a_2 : \tau_2}{A \vdash a_1\ a_2 : \tau_1}$$

$$\text{(Gen-Let)}\ \frac{A \vdash a_1 : \tau_1 \qquad A \oplus (x \mapsto \mathrm{Gen}(\tau_1, A)) \vdash a_2 : \tau_2}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

L'expression $\mathrm{Gen}(\tau, A)$ est la généralisation du type τ dans l'environnement A. C'est le schéma de types obtenu en quantifiant toutes les variables qui sont libres dans τ mais pas dans A.

FIG. 1.1: Règles de typage pour le noyau ML.

Théorème 1 (Auto-réduction) *Les typages sont préservés par réduction (la relation $\subset$ est une sous-relation de $\longrightarrow$).*

Théorème 2 (Progression) *Les expressions bien typées qui ne sont pas des valeurs peuvent être réduites.*

Ces deux théorèmes dépendent évidemment du choix des δ-règles. Ils sont valides pourvu que les deux hypothèses suivantes soient satisfaites.

1. **Auto-réduction** δ est une sous relation de $\subset$,
2. **Progression** Une expression bien typée de la forme $f^k\ v_1 \ldots v_k$ peut toujours se réduire par une δ-règle.

Une autre propriété importante pour le langage ML est l'existence de types principaux et d'un algorithme qui les calcule. Les systèmes de typage que nous considérons vérifient tous le lemme suivant.

Lemme 1 (Stabilité par substitution) *Si $A \vdash a : \tau$ est un jugement valide, alors pour toute substitution θ le jugement $\theta(A) \vdash a : \theta(\tau)$ est aussi valide.*

Cela permet de considérer la synthèse des types comme un problème d'unification et de profiter des formalismes et des outils développés dans le domaine de l'unification pour le résoudre (voir [102]). Un problème de typage est la donnée d'un environnement de typage A, d'un programme a et d'un type τ et consiste en la recherche de l'ensemble des substitutions θ telles que le jugement $\theta(A) \vdash a : \theta(\tau)$ soit valide. La stabilité par substitution implique que l'ensemble des solutions d'un problème de typage est fermé par composition avec une substitution arbitraire. L'existence de types principaux se ramène alors l'existence de solutions minimales aux problèmes de typage.

Théorème 3 (Types principaux) *Un problème de typage qui admet une solution admet une solution minimale. Il existe un algorithme qui étant donné un problème de typage retourne une solution minimale ou un échec si le problème de typage n'a pas de solution.*

Sortes et théorie équationnelle Les résultats s'étendent également lorsque l'algèbre des types est sortée ou munie d'une théorie équationnelle régulière (les deux membres d'une équation possèdent toujours le même ensemble de variables). L'existence de types principaux dépend alors de l'existence d'unificateurs principaux dans l'algèbre considéré. Cette extension est étudiée dans [102]. On pourra se reporter à [58] pour une autre étude dans le cas des théories équationnelles non régulières. La correction du typage pour l'évaluation dépend alors de conditions supplémentaires entre les types des primitives et leur domaine définition.

Types récursifs Les résultats s'étendent également au cas des types récursifs dans l'algèbre libre. Toutefois, en présence d'équations, il faudra prendre garde à ce que les types récursifs commutent avec les équations. Nous traitons seulement un cas particulier dans le chapitre 5 (et nous ne donnons que des résultats purement syntaxiques).

Des variations sur les définitions Une solution d'un problème de typage peut donc instancier les variables libres du contexte de typage autant que les variables du type attendu pour l'expression typée. C'est en ce sens (1) que nous assurons l'existence de solutions principales. Les types principaux sont quelques fois énoncés en interdisant d'instantier les variables libres dans le contexte (2). Lorsque les types sont les termes d'une algèbre libre, il n'y a pas de différence entre les deux résultats. En effet si la solution principale du problème (1) restreinte aux variables du contexte de typage est un renommage, alors on en déduit une solution principale de (1) ; sinon il n'y a pas de solution au problème (2). (La réciproque est facile.) Cette propriété n'est plus nécessairement vraie en présence d'une théorie équationnelle.

Constructions impératives Les résultats présentés dans cette partie s'étendent aux constructions impératives lorsque la construction de liaison `let` $x = a$ `in` a est remplacée par la forme plus restreinte `let` $x = v$ `in` a. En fait, il est possible de relâcher cette condition, et il suffit que v soit une expression non expansive, c'est-à-dire qui s'évalue sans créer d'effet de bord. On peut aussi, de façon encore plus souple permettre de lier une expression a quelconque, mais interdire la généralisation des variables de types qui apparaissent dans le typage d'une sous-expression de a qui n'est pas elle-même une sous expression d'une expression non-expansive de a.

1.2 Les enregistrements polymorphes

Le langage ML peut être facilement étendu avec des enregistrements déclarés, analogues aux types concrets. Ces enregistrements ne sont pas suffisamment puissants pour simuler les objets. En particulier, il n'est pas possible d'écrire une fonction polymorphe qui puisse accéder au même champ ℓ de deux enregistrements de types différents, mais ayant tous deux le même champ ℓ. Le langage SML propose une variante des enregistrements déclarés autorisant la surcharge statique des étiquettes, mais la restriction à de la surcharge statique ne permet pas d'augmenter réellement l'expressivité du langage et revient à une simple convenance de notation.

Les enregistrements polymorphes sont au coeur du typage des objets, comme l'ont très tôt remarqué Cardelli [20], puis Wand [119] dans le cadre du langage ML.

Le typage des enregistrements s'appuie sur les types-enregistrements et le polymorphisme des variables de rangée proposé par Wand [119]. Élaboré précisément dans mes travaux de thèse [100], sa présentation a été ensuite simplifiée et son étude poursuivie dans divers articles : les résultats les plus techniques sont présentés dans [104, 102] et appliqués à une proposition concrète présentée dans le chapitre 2 qui reproduit [106]. Toutefois, nous montrons dans le chapitre 3 que le typage des enregistrements polymorphes peut se décomposer en deux fonctionnalités orthogonales :

- les enregistrements partiels de domaine fini,
- les enregistrements totaux (constants presque partout) de domaine infini.

Cette décomposition permet une présentation plus simple des enregistrements ; elle est aussi enrichissante, car chacun des mécanismes peut être utilisé indépendemment de l'autre. Aussi c'est la présentation que nous suivrons ici. Cette décomposition n'est pas apparue immédiatement, bien que notre premier travail, présenté dans [99], ne couvrait techniquement que le premier aspect. Les

idées pour traiter les domaines infinis étaient présentes, mais pas encore tout à fait matures. Cette décomposition est aussi intéressante parce que le premier aspect peut s'exprimer entièrement dans le langage ML, alors que le second demande une extension significative du langage.

Plusieurs variantes du calcul sur les enregistrements ont été proposées, en général pour éviter des difficultés de typages. Notre séparation en deux fonctionnalités orthogonales permet de les retrouver presque toutes en gardant le même moteur des enregistrements totaux et en modifiant seulement les opérations sur les champs des enregistrements partiels, ou en en ajoutant de nouvelles. Par exemple dans le chapitre 3, nous montrons comment ajouter des opérations de retrait et d'échange de champs. Cela confirme également que pour bien comprendre le typage des enregistrements (ou de structures similaires), il est essentielle de les étudier d'abord sur un champ unique (à la rigueur quelques uns), le mécanisme général en découle alors facilement. Cela sera mis à profit dans le chapitre 8.

1.2.1 Enregistrements partiels de domaine fini

Dans cette partie, nous considérons un ensemble fini d'étiquettes. Pour simplifier la présentation nous pouvons nous limiter à deux ou trois étiquettes. Un enregistrement a égal à $\{a = 1\ ;\ b = true\}$ peut être écrit dans le langage ML après avoir effectué la déclaration de type suivante :

`type bar` $=$ {`a : int; b : bool`}

L'expression a a le type `bar`. La phrase ci-dessus est en quelque sorte une déclaration de type abstrait avec une fonction de construction **fun** (x) **fun** (y) $\{a = x\ ;\ b = y\}$ et deux projections **fun** (y) $(y.a)$ et **fun** (y) $(y.b)$ pour fonctions d'élimination. L'exemple peut être généralisé en déclarant le type `bar` polymorphe :

`type` $(\alpha,\ \beta)$ `bar` $=$ {`a :` α`; b :` β}

Mais un enregistrement b défini seulement sur le champ a est nécessairement d'un autre type `bar'`, préalablement déclaré, différent et incompatible avec `bar`. Le problème du typage des enregistrements apparaît dès que l'on souhaite écrire une fonction d'accès polymorphe, c'est-à-dire une fonction d'accès à un champ ℓ fixé qui puisse prendre en argument tout enregistrement ayant au moins le champ ℓ défini, indépendamment des autres champs.

En l'absence d'enregistrements polymorphes, une solution naïve en ML consisterait à utiliser un enregistrement suffisamment grand possédant tous les champs nécessaires et à ne pas remplir certains champs. Cela se traduit en ML par l'utilisation de valeurs du type suivant :

`type` α `option = Some of` α | `None;;`

On pourrait ainsi écrire {`a = Some 1; b = Some true`} et {`a = Some 0; b = None`} pour représenter respectivement les enregistrements {`a = 1; b = true`} et {`a = 0`}. Cependant cette solution oblige à exécuter un test dynamiquement à chaque accès à un champ de l'enregistrement : le typage ne rend pas compte de la présence ou l'absence d'un champ. Nous avons donc aussi le risque d'une erreur dynamique, alors que nous souhaitons détecter l'absence d'un champ statiquement.

En fait, il existe une solution très simple en ML pour distinguer par le typage si une valeur est présente ou pas. Elle consiste à remplacer le type optionnel par deux déclarations de types indépendantes :

`type` α `pre = Pre of` α `and  abs = Abs;;`

Chacun des constructeurs appartenant alors à un type différent, il est possible de distinguer statiquement les deux cas. Plus précisément, chacun des constructeurs est superflu (ici car unique dans

son type), et pourrait être omis. En tous cas, même s'il reste présent dynamiquement, le test ne peut jamais échoué, et en fait n'est pas effectué. Reprenons le codage de façon plus systématique (nous en profitons pour ajouter un champ supplémentaire) :

```
type (α, β, γ) record = {a : α; b : β; c : γ};;
let get (Pre x) = x;;
let a_ x = get x.a and b_ x = get x.b;;
val a_ : (α pre, β, γ) record → α = ⟨fun⟩
val b_ : (α, β pre, γ) record → β = ⟨fun⟩
```

Nous pouvons écrire le programme suivant en toute sécurité :

```
let x = {a = Pre 1; b = Pre true; c = Abs};;
let y = {a = Pre 0; b = Abs; c = Abs};;
(a_ x) + (a_ y);;
```

L'accès à un champ non défini sera détecté statiquement.

```
b_ y;;
Characters 3–4:
This expression has type (int pre, abs, abs) record
but is here used with type (int pre, α pre, β) record
```

L'extension d'un enregistrement donné avec un nouveau champ peut également être définie :

```
let _a r x = {a = Pre x; b = r.b; c=r.c} and _b r x = {a = r.a; b = Pre x; c=r.c};;
val _a : (α, β, γ) record → δ → (δ pre, β, γ) record = ⟨fun⟩
val _b : (α, β, γ) record → δ → (α, δ pre, γ) record = ⟨fun⟩
```

Nous vérifions sur un exemple que l'extension d'un champ est libre, c'est-à-dire qu'elle peut se faire que le champ soit ou non défini (cela se voit également sur le type de _a ci-dessus).

```
let x' = _a x false and y' = _b y false;;
```

Le codage que nous venons de présenter peut être évité en utilisant des enregistrements extensibles primitifs tout en conservant le mécanisme de typage. Nous avons alors deux applications possibles :

- Utiliser le codage tel quel et sans extension du langage lorsqu'une situation nécessite des enregistrements partiellement définis.
- Ajouter du sucre syntaxique ou, mieux, en déduire des constructions primitives pour les enregistrements polymorphes extensibles de domaine fini pré-défini (c'est en fait la solution exposé dans [99]).

Toutefois, l'une ou l'autre ne convient que pour un usage peu fréquent ou avec des constructions primitives pour de petits ensembles d'étiquettes, et un programme complet. En effet, on ne peut longtemps cacher le problème de la taille des types enregistrements proportionnels au nombre total d'étiquettes apparaissant dans le programme. Lorsque celui-ci devient trop grand et lorsque les enregistrements sont définis sur un très petit sous-ensemble d'étiquettes, les types ont une taille démesurée par rapport aux valeurs et deviennent du bruit illisible.

Pire, cela exclut un ensemble potentiellement infini d'étiquettes, puisque les étiquettes ne sont pas connues à l'avance. Pour des raisons de modularité, il est important de ne pas fixer l'ensemble des étiquettes en fonction d'un programme particulier, mais de se donner dès le départ un ensemble potentiellement infini dénombrable de toutes les étiquettes possibles.

Pour bien expliquer la modularité ou le mécanisme qui permet de ne retenir que les étiquettes significatives, il faut inévitablement considérer un ensemble infini d'étiquettes. L'intuition facile que le cas infini est la limite du cas fini est tout à fait informelle et ne dispense en rien d'une étude rigoureuse.

Ce codage a été repris, avec les limitations que nous venons d'exposer dans [121] dans le cadre de ML et [24] dans un système de types de second ordre. D'autres études plus récentes sur les codages dans le π-calcul suivent une démarche analogue.

Pour aller plus loin, et finaliser complétement l'ajout de constructions primitives, il est souhaitable d'ajouter des sortes aux expressions de types de façon à distinguer les types des champs `P` τ et `A` ainsi que les variables de champs des types usuels. Pour cela, on utilisera une signature très simple qui est l'une des deux signatures présentées dans le chapitre 2.

1.2.2 Enregistrements totaux de domaine infini

Nous résolvons ici le problème orthogonal. Il ne s'agit plus de savoir si un champ est défini ou non, puisque nous supposons tous les champs définis, mais de connaître le type des valeurs portées par tels ou tels champs.

Nous nous limitons ici à des enregistrements totaux presque partout constants. C'est-à-dire que les projections sont toutes égales sauf au plus celles d'un nombre fini de champs. Intuitivement, une valeur d'un enregistrement peut être représentée par ses valeurs particulières sur les champs significatifs et une valeur par défaut sur les autres champs.

$$\{\ell_1 = v_1\ ;\ldots \ell_n = v_n\ ;\ \tau\}$$

Il est donc naturel de typer un tel enregistrement par une expression de type analogue :

$$\{\ell_1 = \tau_1\ ;\ldots \ell_n = \tau_n\ ;\ \tau\}$$

Toutefois, nous apportons quelques corrections : Pour éviter un symbole $\{\_\}$ d'arité variable, il est préférable de voir le type précédent comme $\{(\ell_1\colon \tau_1\ ;\ (\ldots \ell_n\colon \tau_n\ ;\ \tau)\}$ qu'il faut lire

$$\{\}\Big(\ell_1\,(\tau_1, (\ldots \ell_n(\tau_n, \tau)\ldots))\Big)$$

C'est-à-dire que $\{\_\}$ est un constructeur d'arité 1 et $(\ell : \_; \_)$ est un constructeur d'arité deux pour tout $\ell \in \mathcal{L}$. Les expressions $(\ell : \tau; \tau)$ sont dites *expressions de rangée* et ne peuvent pas être utilisées en dehors des types enregistrements. Pour distinguer les types usuels τ des expressions de rangée ρ nous introduisons un nouveau symbole $\langle\_\rangle$ et nous écrirons $\langle\tau\rangle$ pour utiliser τ comme une rangée. Finalement, nous écrivons $\{\ell_1 = \tau_1\ ;\ (\ldots \ell_n = \tau_n\ ;\ [\tau])\}$. La grammaire des expressions de type est :

$$\tau ::= \alpha \mid \tau \to \tau \mid \{\rho\} \qquad\qquad \rho ::= \varphi \mid (\ell : \tau; \tau) \mid \langle\tau\rangle$$

Remarquons que nous avons ajouté des variables de rangée φ qui s'avèrent jouer un rôle essentiel. Pour retrouver l'idée intuitive que l'utilisation d'étiquettes permet de ne plus tenir compte de l'ordre, les rangées sont considérées modulo une équation de commutativité gauche, *i.e.*

$$(\ell_1 : \alpha_1; (\ell_2; \alpha_2; \alpha)) = (\ell_2 : \alpha_2; (\ell_1 : \alpha_1; \alpha)) \qquad \forall \ell_1, \ell_2 \in \mathcal{L}$$

Enfin, l'autre intuition que le type en fin de rangée représente le type des projections sur tous les champs non significatifs est rendue par une équation d'idempotence :

$$\langle\tau\rangle = (\ell : \tau; \langle\tau\rangle) \qquad \forall \ell \in \mathcal{L}$$

Enfin, certaines expressions de type telles que $\{\ell\colon\alpha\ ;\ \ell\colon\alpha'\ ;\ \theta\} \to \{\theta\}$ sont mal formées. D'une part, le champ ℓ est répété deux fois : quel valeur faut-il lui donner ? Il serait possible de choisir la valeur la plus à gauche, comme cela est fait dans [8, 66], mais au prix d'une complication inutile. D'autre part la variable θ apparaît dans deux contextes différents n'étant pas précédés des mêmes ensembles d'étiquettes explicitement définies. L'élimination du cas précédent nous oblige à éliminer ce second cas de façon à ce que l'instance d'un type bien formé soit toujours bien formé. Les expressions de type mal formées peuvent être éliminées par des sortes (se reporter au chapitre 2).

Il est maintenant facile de définir les primitives essentielles sur les enregistrements totaux avec leur types :

Primitive	Type	Sémantique intuitive
$(\_/\ell)$	$[\ell\colon\alpha\ ;\ \phi] \to \alpha$	Accès au champ ℓ
$(\_ \parallel [\ell = \_])$	$[\ell\colon\alpha\ ;\ \phi] \to \alpha' \to [\ell\colon\alpha'\ ;\ \phi]$	Modification du champ ℓ
$[\_]$	$\alpha \to [\langle\alpha\rangle]$	Enregistrement constant

L'expression $[v]$ crée un enregistrement partout égal à v. L'expression $v_1[\ell = v_2]$ retourne un enregistrement égal à v_1 sauf sur le champ ℓ où il vaut v_2. L'expression v/ℓ retrouve la valeur de v sur le champ ℓ. Par exemple, `[0]` est l'enregistrement qui vaut `0` sur toutes ses projections et a pour type `[⟨int⟩]`. L'enregistrement `[0] || [b = true]` qui vaut `true` sur le champ `b` et `0` sur tous les autres champs ; il a pour type `[b : bool; ⟨int⟩]`. L'expression `([0] || [b = true]) / b` s'évalue en `0`.

Pour retrouver les enregistrements partiels de domaine infini, il suffit de composer les deux solutions précédentes. On obtient les primitives décrites dans le tableau ci-dessous :

Primitive	Traduction	Type
$(\_.\ell)$	`fun` (x) `get` (x/ℓ)	$[\ell\colon\mathtt{P}\ \alpha\ ;\ \alpha'] \to \alpha$
$(\_ \parallel \{\ell = \_\})$	`fun` $(r\ x)$ $(r \parallel \{\ell\colon\mathtt{Pre}\ x\})$	$[\ell\colon\mathtt{P}\ \alpha\ ;\ \alpha'] \to \alpha'' \to [\ell\colon\mathtt{P}\ \alpha''\ ;\ \alpha']$
$\{\}$	`[Abs]`	$[\langle\mathtt{A}\rangle]$

Ce tableau peut à nouveau se lire de deux manières différentes, selon la vision (style de programmation ou nouvelles primitives) des enregistrements partiels. La première lecture est celle retenue dans le chapitre 3. La seconde est plus dans l'esprit du chapitre 2. Les types des enregistrements partiels de domaine infini sont restreints par une double signature, chacune correspondant à une des extensions.

Une extension naturelle La version donnée ici des enregistrements totaux est une simplification de celle étudiée dans le chapitre 3. En particulier, les modèles sont ici des variables ou des types usuels $\langle\tau\rangle$. Pour atteindre toute leur puissance, la structure des types est hiérarchisée et inclue une copie des types usuels dans les rangées. Cette extension n'est pas nécessaire pour les enregistrements polymorphes classiques mais elle l'est pour la variante utilisée dans le chapitre 5.

Types étiquetés Les types-étiquetés de Berthomieu, reformalisés ensuite par Le Monier de Sagazan [8] sont une autre solution aux enregistrements totaux de domaine infini. On en déduit

immédiatement une autre solution au typage des enregistrements polymorphes partiels de domaine infini par composition avec notre solution pour les enregistrements partiels. Dans leur version de base, les types-étiquetés et les types-enregistrements ont même expressivité, mais ils diffèrent rapidement dès que l'on en considère des extensions. Nous pensons aussi que les types enregistrements sont plus simples parce qu'ils rentrent dans le formalisme bien compris des algèbres sortées et des théories équationnelles et ne nécessitent pas de modifier les règles de typage de ML.

Autres applications Les enregistrements totaux sont des enregistrements avec valeurs par défaut. Leur intérêt ne se limite pas à cette application immédiate. Ils sont aussi une forme duale des variantes : en effet les variantes ont naturellement un cas d'action par défaut pouvant s'appliquer à n'importe quel constructeur (portant une valeur du bon type). Ils permettent également de décrire des types plus précis. Cela n'est pas sans importance car cette extension sera indispensable pour définir les types des objets et des messages dans le langage ML-ART.

Polymorphisme et variables de rangée Dans notre approche du typage des enregistrements, les opérations de sélection et d'extension reposent uniquement sur le polymorphisme (des variables de rangées), ce qui évite toute perte d'information de type. La finesse des types-enregistrements permet de les manipuler exactement. Il est possible d'extraire l'information de types sur tous les champs sauf un nombre fini pour reconstruire un nouvel enregistrement, avec des types différents sur ces champs particuliers, mais exactement le même comportement sur les autres champs, qu'ils soient définis ou non. Ce sont ces deux aspects, tous deux absents dans l'approche plus fréquente avec du sous-typage, qui nous permettent de rester au premier ordre, donc de conserver la simplicité du langage ML pour typer précisément et simplement les enregistrements, mais aussi et c'est sans doute encore plus important, les objets dans les chapitres 5 et 6. Les types-enregistrements possèdent des « destructeurs de types » dans le sens de Hofmann et Pierce [53]. Toutefois, les types enregistrements n'offrent que l'opération très simple d'expansion. La contraction, plus compliquée, étudiée dans [100], est en fait un raffinement qui n'est pas vraiment nécessaire. L'ajout de sous-typage est possible, mais il est secondaire, et permet simplement de confondre des enregistrements de domaines différents, en cachant le sous-domaine sur lequel ils diffèrent.

1.3 Concaténation des enregistrements

Le problème de la concaténation des enregistrements se pose naturellement comme une généralisation de l'opération d'extension. En pratique son importance est justifiée par son analogie avec l'héritage. Alors que l'extension des enregistrements permet de simuler l'ajout de méthodes à un objet existant donc la notion d'héritage simple, la concaténation des enregistrements correspond naturellement à l'héritage multiple.

Le problème de la concaténation est celui de donner un type suffisamment général à la primitive $\|$ qui prend deux enregistrements et en retourne un troisième composé de tous les champs du second plus ceux qui sont dans le premier mais pas dans le second. La seule contrainte sur les arguments est qu'ils soient des enregistrements, donc de types $\{\alpha\}$ et $\{\alpha'\}$. Mais comment exprimer le type du résultat ? Intuitivement, nous souhaiterions écrire $\{\alpha \parallel \alpha'\}$ pour les types des champs pris par priorité dans α' et dans α lorsqu'ils ne sont pas définis (*i.e.* de type `A`) dans α'. Pour formaliser cette proposition, il faudrait introduire un constructeur $\|$ avec des équations et une stratégie de résolution pour étendre les algorithmes d'unification sur de tels termes. La plupart des tentatives pour résoudre ce problèmes ont échoué, au moins partiellement. Les premières sont restreintes et compliquées [50, 27]. Une proposition récente très simple change la sémantique des enregistrements

et la rend dépendante des types [125]. D'autres propositions utilisent une forme de types intersection et sont plus coûteuses [121]. Nous avons également étudié la concaténation à partir d'un mécanisme de sous-typage dans [110] qui ressemble à une forme dégradée de types intersection.

Nous proposons ici une solution opposée qui est plutôt un contournement de la difficulté que sa résolution effective. Elle consiste à considérer (donc à typer) un enregistrement r comme une fonction $\texttt{fun}\ (u)\ (u \parallel r)$ qui ajoute ses propres champs à un enregistrement u arbitraire reçu en argument. L'intérêt de cette construction provient du fait que l'enregistrement r est toujours connu, seul l'argument u restant arbitraire. En s'appuyant sur la composition, on ramène ainsi la concaténation de deux enregistrements arbitraires à la concaténation à gauche, c'est-à-dire à une opération d'extension.

Nous donnons ci-dessous les primitives sur les enregistrements avec concaténation, leur codage en terme d'enregistrements avec extension, et le type des expressions primitives et codées.

Concaténation (primitive)	Traduction
$(\_.\ell)$ $\alpha \to \{\ell\colon \alpha'' \Rightarrow \mathtt{P}\ \alpha'\ ;\ \alpha\}$	$\texttt{fun}\ (x)\ \texttt{fun}\ (u)\ (u \parallel \{\ell = x\})$ $\alpha \to \{\ell\colon \alpha''\ ;\ \alpha\} \to \{\ell\colon \mathtt{P}\ \alpha'\ ;\ \alpha\}$
$\{\}$ $\{\alpha \Rightarrow \alpha\}$	$\texttt{fun}\ (u)\ u$ $\{\alpha\} \to \{\alpha\}$
$(\_ \parallel \_)$ $\{\alpha \Rightarrow \alpha'\} \to \{\alpha' \Rightarrow \alpha''\} \to \{\alpha \Rightarrow \alpha''\}$	$\texttt{fun}\ (r\ r')\ \texttt{fun}\ (u)\ (u \parallel r) \parallel r'$ $(\{\alpha\} \to \{\alpha'\}) \to (\{\alpha'\} \to \{\alpha''\}) \to \{\alpha\} \to \{\alpha''\}$

Le constructeur de type $\Rightarrow$ utilisé ici est distributif par rapport aux symboles $(\ell : \_; \_)$.

Au sens strict, cette proposition est d'abord un style de programmation qui permet d'éviter le problème de la concaténation des enregistrements. C'est aussi un système de typage pour la concaténation des enregistrements permettant la synthèse des types. Plus généralement, c'est une méthode pour étendre un système de typage avec une opération d'extension à un système plus riche avec une opération de concaténation.

1.4 Les objets vus comme des enregistrements

Le lien entre les objets et les enregistrements a été mis en évidence par L. Cardelli dès 1984 [20] et repris par M. Wand [121] dans le contexte de ML en 1988. Toutefois, une correspondance plus étroite et typée n'a été réalisée que vers 1994 [93] par B. Pierce et D. N. Turner. Dans le chapitre 5 présenté aussi dans [108], nous reprenons cette démarche, mais avec deux contraintes supplémentaires. Nous nous imposons une efficacité et une concision du code source qui soient réalistes pour une utilisation dans un vrai langage; d'autre part nous choisissons le langage ML ce qui nous oblige à synthétiser tous les types, donc à rester pour l'essentiel au premier ordre.

Les objets

Dans leur version la plus simple, les objets sont des enregistrements de valeurs. Afin de distinguer les objects des autres enregistrements, il serait naturel de définir le type suivant[1].

[1] Les exemples de cette partie sont typables dans une maquette du langage ML-ART.

```
type (α) objet = Objet of {α}
```

Toutefois cette notation obligerait à expliquer que le paramètre α du type `objet` est une variable de rangée et donc à expliquer les sortes à l'utilisateur. Pour éviter ce problème, mais aussi pour rendre les types des objets plus lisibles, nous préférons utiliser une extension de la notion d'abréviation de type introduite par B. Berthomieu dans LCS [7]. Celle-ci permet l'utilisation du filtrage dans les expressions de type en paramètre. Par exemple, on pourra écrire :

```
type (α * β) fst = α;;
```

Un telle définition impose que l'argument du constructeur `fst` soit filtré par le motif de sa définition, c'est-à-dire ici une expression de type paire. Cette forme d'abréviation est d'autant plus pratique qu'elle est compatible avec l'égalité sur les types enregistrements. Nous écrirons plutôt :

```
type ({α}) object = Object of {α};;
```

Dans cette vision simplifiée, l'envoi de message se réduirait à l'accès au champ correspondant dans l'enregistrement.

```
let send_m (Object p) = p.m;;
```

Séparer les variables d'instance des méthodes

En fait, on distingue dans un objet les variables d'instance qui sont des champs d'enregistrements comme décrits ci-dessus, et des méthodes qui sont des procédures qu'il faut exécuter à l'invocation d'un message. Pour plusieurs raisons, il est important de distinguer les variables d'instance des méthodes. D'une part, il est fréquent qu'un objet possède quelques variables d'instance et de nombreuses méthodes. De plus, de nombreux objets ont les mêmes méthodes et ne diffèrent que par leurs variables d'instance. L'enregistrement des méthodes peut alors être partagé.

Pour renforcer cette distinction, et simplifier la présentation, nous considérons un objet comme une paire composée d'un état interne R (les variables d'instance) et d'un vecteur de méthodes M. Les méthodes doivent pouvoir consulter l'état interne de l'objet. Pour rendre les méthodes indépendantes de l'état interne R (ce qui permettra ensuite d'en hériter), il est préférable de les abstraire systématiquement par rapport à R

Avant de donner le type des objets, précisons que la structure des types enregistrements de ML-ART est une variante de ceux décrits ci-dessus (elle est présentée plus en détail dans le chapitre 2) :

$$\varphi ::= \theta \mid \mathtt{P}\ \tau \mid \mathtt{A} \qquad \text{est remplacé par} \qquad \varphi ::= \theta \mid \eta.\tau \quad \text{où} \quad \eta ::= \epsilon \mid \mathtt{P} \mid \mathtt{A}$$

Ainsi, au lieu de dire qu'un champ est absent ou présent avec un certain type, nous décrivons séparément le type possible (au sens usuel) des valeurs d'un champ, qu'il soit présent ou absent, et sa présence. Par exemple, l'enregistrement a égal à $\{\ell_1 = 1\ ;\ \ell_2 = true\}$ a maintenant pour type $\{\ell_1{:}\mathtt{P}\ .\mathtt{int}\ ;\ \ell_2{:}\mathtt{P}\ .\mathtt{bool}\ ;\ \mathtt{A}.\alpha\}$. Cette variante présentée à la fin du chapitre 2 est strictement plus expressive. Il est facile de traduire les types précédents dans cette forme enrichie. Cette structure permet aussi de manipuler le type d'un enregistrement de fonctions `roff` (for *record of functions*) comme celui d'une fonction retournant un enregistrement.

```
type  ({'arg → ('attendance. 'methods)}) roff = {'attendance. 'arg → 'methods};;
```

Nous « jouons » avec les équations de distribution des types enregistrements pour en donner une lecture plus concise : lorsque le type `'arg` est identique dans tous les champs de l'enregistrement, l'écriture de gauche permet de ne le mentionner qu'une seule fois.

Cette notation permet de définir le type des méthodes comme suit :

```
type ('R, {'I}) objectM == {['R] → 'I} roff;;
```

Ainsi, les méthodes `{['R] → 'I} roff` sont un enregistrement de fonctions de domaine commun `'R`. En effet, ici `['R]` est un "modèle" constant partout égal à `'R`. De façon plus importante, cette notation permet de séparer le type commun `'R` de l'état interne des objets de celui de leur interface `{'I}`, ce qui permettra ensuite d'abstraire l'état interne.

```
type ('R, {'I}) objectRM == ('R * ('R, {'I}) objectM);;
```

Cacher l'état interne

Dans de nombreuses situations, les variables d'instance déterminent la représentation de l'objet qui est souvent un choix d'implantation dont ne dépendent que les méthodes de l'objet. Inversement les méthodes déterminent l'interface de l'objet qui est utilisée par les autres parties du programme. Il est fréquent de vouloir cacher la représentation de l'objet, que ce soit pour des raisons de sécurité, de lisibilité ou simplement pour permettre à des objets ayant des implantations différentes mais des interfaces identiques d'être échangés. Un exemple classique est celui des points dans le plan que l'on peut représenter en coordonnées polaires ou en coordonnées cartésiennes selon les opérations que l'on souhaite privilégier.

Pour cacher l'état interne, nous reprenons la solution B. Pierce et D. Turner [93]. Pour cela, il suffit ajouter des types existentiels au langage, par exemple en reprenant la proposition de K. Laüfer et M. Odersky [64, 65]. On peut alors abstraire l'état interne `'R` dans le type des objets

```
type ({'I}) object =  Object of Exist ('R)  ('R, {'I}) objectRM;;
```

Permettre les appels récursifs

Dans la programmation avec objets, il est important qu'une méthode puisse envoyer récursivement d'autres messages à l'objet qui l'a invoquée. Dans [93]. B. Pierce et D. N. Turner résolvent ce problème en définissant les méthodes d'un objet récursivement. Cependant, en raison de la difficulté à définir un point fixe sur les enregistrements avec une stratégie d'évaluation en appel par valeur, leur codage ne s'applique que dans un langage avec une stratégie d'évaluation en appel par nom. Il en résulte que chaque appel récursif réévalue, donc recopie, l'enregistrement des méthodes. Pour éviter cette source d'inefficacité (et d'autres problèmes), nous préférons utiliser le mécanisme dit d'auto-application. Il consiste à abstraire les méthodes par rapport à `R` et `M` plutôt que par rapport à `R` seul. En contrepartie, le type des méthodes devient récursif :

```
type ('R, {'I}) objectM == rec 'M in ({['R * 'M] → 'I}) roff;;
```

En effet, les méthodes qui composent un objet ont pour domaine le type de l'objet lui-même. En revanche l'enregistrement des méthodes lui-même n'est plus récursif (à l'inverse de la proposition précédente où les objets étaient récursifs, mais pas leurs types). Nous reprenons alors les définitions précédentes inchangées pour obtenir le type final des objets dans ML-ART :

```
type ('R, {'I}) objectRM == ('R * ('R, {'I}) objectM);;
type ({'I}) object =  Object of Exist ('R) ('R, {'I}) objectRM;;
```

L'état interne caché à l'extérieur de l'objet, reste visible par les méthodes. Pour cela, il est important que la récursion soit à l'intérieur de l'abstraction, et non l'inverse. En particulier, les méthodes sont définies en ayant une vision exacte de l'état interne `'R`, ce qui leur permet d'accéder ou de modifier l'état interne. Bien sûr, on aurait pu écrire directement :

```
type ({'I}) object = Object of Exist ('R) rec 'RM in ('R * ({['RM] → 'I}) roff)
```

Définir des messages polymorphes

Lorsqu'un message est utilisé à l'intérieur d'un objet, il interagit avec l'état interne puisqu'il sélectionne une méthode (donc une fonction) qui reçoit l'état interne de l'objet comme argument. Si un tel message est importé de façon monomorphe, il en résulte une extrusion de l'état interne à l'extérieur de l'objet par l'intermédiaire du message. Les fonctions importées de l'extérieur pour être appliquées à une valeur comportant des parties cachées doivent être suffisamment polymorphes pour laisser les parties cachées bien cachées... Cette observation, qui semble ne pas avoir été remarquée auparavant, est pourtant générale : la présence de types existentiels n'a guère d'intérêt sans l'existence simultanée de types universels. Heureusement, ces derniers peuvent être rattachés à des déclarations de types concrets comme les types existentiels tout en étant techniquement plus simples que ceux-ci. Le type des messages est :

```
type ({'I}, α) message = Message of
    All ('R)  ('R, {'I}) objectM → ('R, {'I}) objectRM → α;;
```

Les messages sont polymorphes par rapport à la représentation interne des objets. L'envoi de messages est une fonction uniforme.

```
let repr S = fst S;;
let meth S = snd S;;
let send message P =
  let (Object S) = P in let (Message extractor) = message in extractor (meth S) S;;
```

Reste à définir quelques messages...

```
let setx = Message (fun x → x.setx);;
let getx = Message (fun x → x.getx);;
```

Enfin, un objet !

```
let point =
  let pointR = {mutable abscisse = 0} in
  let pointM =
    let getx self   = (repr self).abscisse
    and setx self x = (repr self).abscisse ← x
    and move self x = send setx (Object self) (x + send getx (Object self))
    in emptyM || {getx; setx; move}
  in Object (pointR, pointM);;
```

La valeur `emptyM` est simplement l'enregistrement vide mais avec le type $(\alpha \to \{\texttt{abs. null}\})$ `objectM` où `null` est un nouveau symbole de type, utilisé simplement pour rendre les objets monomorphes.

Les classes

Comme nous l'avons rappelé, de nombreux objets ne diffèrent que par leurs variables d'instance. En fait, les objets sont en général construits à partir de classes. Une classe doit permettre deux opérations bien distinctes : générer des objets de cette classe et créer d'autres classes par héritage. En particulier, une classe est abstraite par rapport aux variables d'instance. Puisque les méthodes ne dépendent pas des variables d'instance, il suffit d'abstraire l'état interne par rapport à celles-ci. Une première tentative est donc est de représenter une classe par la paire d'une fonction pour

créer l'état interne, et d'un enregistrement de méthodes. Pour hériter d'une classe, il suffirait de lui ajouter les variables d'instance et les méthodes supplémentaires de la classe héritée.

Toutefois, il est plus facile de prendre en compte directement l'héritage multiple. Par analogie avec la concaténation des enregistrements, l'enregistrement des méthodes `M` d'une classe `C` est abstrait par rapport aux méthodes `superM` d'une classe parente hypothétique à laquelle la classe `C` pourrait ultérieurement être ajoutée. De même, l'état interne `R` de la classe `C` est abstrait par rapport au constructeur de l'état interne `superR` de la classe parente. Concrètement, la classe des points sera définie ainsi :

```
let pointC =
  let pointR superR v = superR || {mutable abscisse = v} in
  let pointM superM =
    let getx self   = (repr self).abscisse
    and setx self x = (repr self).abscisse ← x
    and move self x = send setx (Object self) (x + send getx (Object self))
    and print self  = print_int (send getx (Object self))
    in superM || {getx; setx; move; print}
  in  Class (pointR, pointM);;
```

Le lien entre les classes et les objets est clarifié par la fonction d'instantiation :

```
let new (Class (RR, MM)) v =  Object (RR {} v, MM emptyM);;
```

Le constructeur de type `Class` ci-dessus est utilisé principalement pour imposer une contrainte sur le type des composantes et ainsi anticiper certaines erreurs de types. Sa définition n'est elle-même ni facile ni très lisible, mais le constructeur permet aussi de rendre le type des classes plus lisible, car moins polymorphe. Une définition approchée est :

```
type ('superR → 'init → 'R, {'superI → 'I}) class =  Class of
     ('superR → 'init → 'R) *
     ({ [ ('R, {'I}) objectRM ] → 'superI} roff → ('R, {'I}) objectM)
```

La définition réelle est légèrement différent car il faut dissocier les occurrences de `'R` dans le constructeur de l'état interne et dans l'enregistrement des méthodes.

Le codage des classes par des enrouleurs (*wrappers*) offre directement l'accès aux messages de la super-classe. Pour envoyer un message à la classe parente, il suffit d'extraire la fonction correspondante dans `superM` et de lui passer la vision interne de `self` en argument. Pour illustrer l'héritage multiple, nous définissons une classe couleur :

```
let colorC =
  let colorR superR c = superR || {color = c} in
  let colorM superM = superM || {
    print = fun self → superM.print self; print_string (repr self).color
  } in Class (colorR, colorM);;
```

L'héritage est obtenu par composition des composantes génératrices des classes.

```
let inherit (Class (RR1, MM1)) (Class (RR2, MM2))  =
  let RR superR (v1,v2) = RR2 (RR1 superR v1) v2 in
  let MM superM = MM2 (MM1 superM) in
  Class (RR, MM);;
let colored_pointC() = inherit pointC colorC;;
```

Et voici le point coloré !

```
let p = new (colored_pointC()) (1,"blue");;
send print p;;
1blue- : unit = ()
```

Le langage ML-ART permet de nombreuses autres constructions, plus avancées. Par exemple, il est possible de retourner self exactement, une copie de self, ou une version de self dans laquelle certaines variables d'instance ont été modifiées. Nous verrons ces opérations ci après dans le langage Objective ML.

Notre proposition est avant tout un style de programmation avec objets qui s'appuie de façon essentielle sur les enregistrements polymorphes extensibles. Nous avons pour cela fourni une librairie pour les objets écrite dans le langage ML-ART. Cette librairie utilise toutes les constructions mises au point dans les chapitre 2, 3, 4, les types existentiels de K. Laüfer et M. Odersky [64], les types universels qui sont un complément indispensable aux types existentiels. Nous avons également ajouté des définitions de types avec filtrage, mais seulement pour rendre les expressions de type plus lisibles.

Comparée avec la proposition de B. Pierce and D. N. Turner, notre solution est plus avancée. D'une part elle prend en compte un ensemble plus complet d'opérations sur les objets. Elle est également plus réaliste, dans le sens où le code source est concis et non redondant. On pourrait en fait reprendre le codage de ML-ART dans un calcul avec types explicites d'ordre supérieur. Cela permettrait en outre de résoudre le problème de la lisibilité des types grâce à des opérateurs de types plus puissants.

1.5 Objective ML

L'expérience précédente est satisfaisante sur le plan théorique. En revanche elle est un peu décevante sur le plan pratique. En effet, les types des classes que nous nous sommes bien gardés de montrer deviennent vite très gros, et donc illisibles. La compréhension des erreurs de typage qui non seulement oblige à les lire, mais aussi à les comparer est encore plus difficile. Le problème n'est pas surprenant, et il est lié à la combinaison de deux facteurs aggravants. D'une part la synthèse totale des types signifie aussi l'absence d'abréviations de types. Les types des objets reflètent entièrement leur structure. D'autre part, les objets sont par essence des structures compliquées. Chaque donnée est accompagnée de l'ensemble de toutes les opérations qui peuvent être effectuées sur cette donnée. Dans la vision tout objet, un entier devient une structure comportant une variable d'instance décrivant la valeur de l'entier et autant de méthodes que de fonctions disponibles sur les entiers, certainement plus d'une vingtaine. Ce facteur est encore aggravé par la structure récursive des objets. Cela n'augmente pas nécessairement la taille des types à condition de les représenter (et de les afficher) comme des graphes, mais dans tous les cas, en diminue leur lisibilité. Il devient donc nécessaire d'abréger les types des objets.

C'est la première justification d'Objective ML. Le système des types d'Objective ML est en fait moins expressif que celui de ML-ART. L'exercice consistait à simplifier le système de types de ML-ART autant que possible, pour en simplifier la présentation théorique, mais en conservant tous les exemples pratiques. La plupart des simplifications sont permises par l'utilisation de constructions primitives et par la restriction des classes à être déclarées au niveau le plus haut du langage (*i.e.* elles ne sont pas autorisées sous abstraction). Ces restrictions, en plus des simplifications théoriques ont permis de mettre en place un mécanisme d'abréviation automatique des types très sophistiqué. C'est la seule technique vraiment nouvelle développée pour Objective ML, mais elle est essentielle

et réellement efficace. La plus grande partie du travail reste toutefois l'ajustement et la mise en œuvre harmonieuse de techniques connues. La puissance, la souplesse et la légèreté du langage en dépendent simultanément.

Plutôt que de montrer les détails techniques du mécanisme d'abréviation ou l'étude formelle du langage, nous préférons illustrer le résultat au travers d'une petite démonstration de la couche objet du langage Ocaml [70]. Nous renvoyons le lecteur au chapitre 6 pour une présentation formelle du langage ou du mécanisme d'abréviation.

Montrer la réussite d'un langage avec synthèse des types est une tâche difficile, car plus le système de types est expressif, moins il y a de types à montrer ! Les exemples de cette partie ont tous été exécutés automatiquement dans la boucle interactive du langage Objective Caml (Ocaml en abrégé). Les types ne sont imprimés que lorsqu'ils sont significatifs ou dans la partie 1.5 où nous nous intéressons plus précisément au typage.

De la programmation classique aux objets

Nous en profitons pour rappeler quelques idées qui ne sont pas particulièrement liées au langage Ocaml, mais au style de programmation avec objets. De façon traditionnelle, les données sont distinguées des fonctions qui opèrent sur celles-ci. Les données sont les entiers, les caractères, mais aussi les types concrets :

```
type pierre =  Opal | Perle | Diamant;;
type α liste = Cons of α * α liste | Nil;;
```

Les fonctions sont primitives ou définies sur les types de bases, ou bien définies par filtrage. Ci-dessous la fonction `concasser` séparent des cailloux en leurs éléments constituants :

```
let concasse =  function
    Opal    → ['O'; 'p'; 'a'; 'l'; ' ']
  | Perle   → ['P'; 'e'; 'r'; 'l'; 'e'; ' ' ]
  | Diamant → [ 'D'; 'i'; 'a'; 'm'; 'a'; 'n'; 't'; ' '];;
```

Dans cette vue traditionnelle, le calcul est décrit par l'application d'une fonction à des données

```
let imprime_caillou x = List.iter print_char (concasse x);;
imprime_caillou Opal;;
```

Dans le style à objets, une donnée est regroupée avec l'ensemble de toutes les fonctions qui peuvent opérer sur celle-ci, dans une structure appelée *objet*. Les objets sont construits à partir des classes. Les classes décrivent au travers des méthodes le comportement d'un ensemble d'objets, mais elles abstraient les valeurs particulières qui différencient les objets de la classe.

```
class entier n =
  object
    val valeur = n
    method zéro = (valeur = 0)
    method successeur = {⟨ valeur = valeur + 1 ⟩}
    method prédécesseur = {⟨ valeur = min 0 (valeur−1) ⟩}
    method imprime = print_int valeur
  end;;
```

L'expression {⟨ `valeur = valeur + 1` ⟩} retourne une copie modifiée de self, et ne produit pas d'effet de bord.

```
class caillou roche =
  object
    method concasse = concasse roche
    method imprime = imprime_caillou roche
  end;;
```

Un objet est créé comme instance d'une classe.

```
let treize = new entier 13;;
let pierre = new caillou Diamant;;
```

Le calcul est maintenant décrit pas *envoi de message*.

```
treize # imprime; pierre # imprime;;
13Diamant - : unit = ()
```

Plusieurs messages du même nom (comme `imprime` dans l'exemple ci-dessus) peuvent avoir un comportement différent. Cela est rendu possible parce que le comportement qui doit être exécuté à la réception du message est porté par l'objet qui le reçoit, du moins formellement. Cette aptitude fournit une nouvelle forme de polymorphisme appelée *envoi de messages polymorphes*. Pour illustration, nous définissons une fonction `écho` qui répète l'envoi du message `imprime`.

```
let écho x = x # imprime; x # imprime;;
val écho : < imprime : α; .. > → α = <fun>
```

Cette fonction peut être appliquée à n'importe quel objet possédant une méthode `imprime`.

```
écho treize; écho pierre;;
1313Diamant Diamant - : unit = ()
```

L'envoi de messages polymorphes est une opération essentielle. Un langage avec cette possibilité peut déjà prétendre offrir le "style" de la programmation avec objets.

Héritage et liaison tardive

Un langage à objets doit aussi posséder un mécanisme d'héritage qui permette de construire de nouvelles classes à partir de classes existantes, où éventuellement, de nouveaux objets à partir d'autres objets. La classe des cailloux peut être enrichie en une classe de bijoux, par ajout d'un champ `valeur` et d'une méthode pour calculer le prix à partir de la valeur.

```
class bijou roche valeur =
  object
    inherit caillou roche as pierre
    method prix = 2 * valeur
    method imprime = pierre # imprime;  print_int valeur; print_string " carats"
  end;;
```

Un bijou est imprimé en le considérant d'abord comme une pierre sans valeur (le mot clé `as` lie le bijou à la variable `pierre` avant que les nouvelles méthodes ne soient ajoutées), puis le prix est affiché suivi de l'unité.

```
let solitaire = new bijou Diamant 13;;
solitaire # imprime;;
Diamant 13 carats- : unit = ()
```

Les fonctions récursives jouant un rôle essentiel dans le style de programmation traditionnel, les méthodes doivent elles aussi pouvoir être récursives. Cela est réalisé en général par l'utilisation d'un mot clé `self` dans le corps des méthodes qui réfère à l'objet ayant invoqué la méthode. Nous lui préférons une construction de liaison récursive au niveau des définitions de classes. Nous illustrons la liaison récursive en prenant pour exemple le jeu de la roulette russe.

```
class roulette () =
  object (jeu)
    val mutable position = 0
    method roulette = position ← Random.int 8; jeu
    method coup = position ← (position + 1) mod 8; position = 0
    method je_joue = if jeu # coup then "Je meurs" else jeu # tu_joues
    method tu_joues = if jeu # coup then "Tu meurs" else jeu # je_joue
  end;;
```

Il est important que les méthodes `je_joue` et `tu_joues` s'appellent récursivement pour obliger l'adversaire à jouer quand son tour est venu. Une fois commencé, le jeu ne peut plus s'arrêter que par la mort de l'un des deux joueurs. Sinon, le huitième coup ne serait jamais tiré... Jouons !

```
(new roulette ()) # je_joue;;
- : string = "Tu meurs"
```

Heureusement, c'était un coup à blanc ! Rejouons pour de bon, mais sans oublier de faire tourner la roulette auparavant...

```
(new roulette ()) # roulette # je_joue;;
- : string = "Tu meurs"
```

La capacité pour un objet de s'envoyer récursivement des messages réalise simultanément le mécanisme de la *liaison tardive*. En effet la résolution des appels récursifs n'est pas effectuée au moment de la définition de la classe (comme c'est le cas dans les langages fonctionnels), mais elle est retardée jusqu'au moment de la création de l'objet. Nous illustrons cela en spécialisant la classe de la roulette à un jeu à deux contre un. La classe `roulette_à_deux_contre_un` hérite de la classe `roulette` et redéfinit la méthode `je_joue` pour qu'elle tire un deuxième coup —si le premier n'a pas tué le joueur— avant d'appeler la méthode `tu_joues` comme auparavant.

```
class roulette_à_deux_contre_1 () = object (jeu)
  inherit roulette () as vieux_jeu
  method je_joue = if jeu#coup then "Je meurs" else vieux_jeu # je_joue
end;;

(new  roulette_à_deux_contre_1 ()) # roulette # tu_joues;;
- : string = "Tu meurs"
```

Il est important que la méthode `tu_joues` bien qu'inchangée appelle maintenant la nouvelle méthode `je_joue`, sinon le joueur croyant avoir l'avantage serait trompé.

Une nouvelle forme de modularité

Le style de programmation traditionnel permet de définir de nouvelles fonctions opérant sur une donnée existante de façon modulaire, c'est-à-dire sans avoir à connaître ou à redéfinir les autres opérations. Par contre, le moindre changement dans la représentation des données oblige à redéfinir toutes les fonctions opérant sur ce type de donnée.

Le style à objets permet d'effectuer les deux types d'extensions, *i.e.* l'ajout d'opération (extension verticale) ou l'extension de la structure de donnée (extension horizontale), de façon modulaire.

Nous illustrons cela sur la structure bien connue de cellules. Par opposition à la classe `caillou` nous choisissons une approche objet plus fine pour les cellules. Chaque type de donnée, *i.e.* chaque constructeur de valeur sera codé par une classe différente, mais ayant la même interface, de façon à pouvoir mélanger toutes les valeurs indépendemment de leur constructeur. Nous groupons ces définitions de classes dans un module, mais c'est seulement pour mieux gérer l'espace des noms.

La classe principale `cons` définit les vraies cellules. L'autre classe définit la cellule vide `nil`. Pour lui donner la même interface que la classe `cons` nous y définissons les fonctions `car` et `cdr` qui lèvent une exception. Cela n'est pas une perte de sécurité, mais la contrepartie d'une exception de filtrage incomplet en Ocaml, par exemple lorsqu'on demande la tête d'une liste vide.

```
module Cell = struct
  class [α, β] cons (h:α) (t:β) =
    object method null = false  method car = h  method cdr = t end

  exception Null
  class [α, β] nil =
    object
      method null = true
      method car = (raise Null : α)
      method cdr = (raise Null : β)
    end
end;;
```

Ces cellules sont très générales. En particulier, il est possible d'affecter des valeurs indépendantes à chacune des cases.

```
new Cell.cons treize solitaire;;
- : (entier, bijou) Cell.cons = ⟨obj⟩
```

Elles diffèrent des paires seulement par la possibilité qu'elles ont d'être non initialisées (la cellule `nil`).

Ne nous privons pas d'un détour amusant par le problème des listes alternées qui ont été proposé par la communauté Java comme un exemple difficile. Pour augmenter la lisibilité, nous définissons une abréviation de type `(α,β) alt_list` pour la structure de liste alternée dont les éléments sont alternativement de type α et de type β.

```
type (α, β) alt_list = (α, (β, (α, β) alt_list) Cell.cons) Cell.cons;;
```

Puis nous définissons un exemple arbitraire de liste alternée.

```
let x = ref (new Cell.nil : (α, β) alt_list) in
  x := new Cell.cons treize (new Cell.cons solitaire (!x)); !x;;
```

Cet exemple générique est compatible avec l'héritage. Les sous-classes des cellules construites par héritage pourront encore être utilisées pour fabriquer des listes alternées (par exemple avec une opération d'itération). La structure de liste traditionnelle se retrouve simplement en forçant le type des cellules à être récursif de période un.

```
module L = struct
  class [α] cons h t = object (self : 'mytype) inherit [α, 'mytype] Cell.cons h t end
  class [α] nil = object (self : 'mytype) inherit [α, 'mytype] Cell.nil  end
end;;
```

Extension verticale Elle est réalisée par l'héritage qui est une opération entièrement modulaire en Ocaml. Le code de la classe héritée ne doit pas être réécrit. De plus, l'héritage multiple permet de définir l'ajout d'une fonctionnalité séparément, puis d'effectuer cet ajout sur différentes classes parentes. Par exemple, on peut définir une palette d'extensions et choisir ultérieurement la bonne combinaison des options pour une application donnée.

Pour ajouter une opération d'itération sur les listes, nous définissons dans un premier temps un module `Iter` qui décrit l'opération à la fois sur les vraies cellules et sur les cellules vides. Dans la classe `cons` les méthodes `car` et `cdr` sont dites virtuelles, parce qu'elles sont utilisées sans avoir été définies. Les méthodes virtuelles d'une classe devront être définies dans une sous-classe, et il n'est pas possible de construire d'objet d'une classe virtuelle.

```
module Iter = struct
  class virtual [α] cons =
    object (self : β)
      method virtual car : α  method virtual cdr : β
      method iter (f : α → unit) = f self#car; self#cdr#iter f; ()
    end
  class [α] nil = object method iter (f : α → unit) = () end
end;;
```

Les listes avec itération sont obtenues par héritage.

```
class [α] cons h t = object (self) inherit [α] L.cons h t  inherit [α] Iter.cons end
class [α] nil = object inherit [α] L.nil inherit [α] Iter.nil end;;
```

Comme test, nous pouvons construire une liste de nombres premiers et l'imprimer.

```
let primes =  List.fold_right (new cons) [2;3;5;7;11;13] (new nil);;
val primes : int cons = ⟨obj⟩
primes#iter print_int;;
23571113- : unit = ()
```

L'extension verticale consiste à ajouter un constructeur à un type de donnée existant. Supposons par exemple que les listes soient souvent concaténées. Pour éviter le calcul de la concaténation il est possible, dans le style traditionnel, d'ajouter un nouveau constructeur `Append` au type des listes. Cependant cela crée un nouveau type, incompatible avec le précédent et il faut en conséquence redéfinir toutes les opérations sur les listes.

Avec des objets, il suffit de définir une nouvelle classe `append` qui décrit le comportement du nouveau constructeur. Les anciens constructeurs restent compatibles et n'ont donc pas besoin d'être modifiés.

```
class [α] append l r = object (self : 'mytype)
  val left = (l : 'mytype)  val right = (r : 'mytype)
  method null = left#null && right#null
  method car = if left#null then right#car else (left#car : α)
  method cdr =
    if left#null then right#cdr
    else if left#cdr#null then right else {⟨ left = left#cdr ⟩}
  method iter (f : α → unit) = left#iter f; right#iter f; ()
end;;
```

Nous introduisons une notation infixée, et vérifions le comportement sur un exemple arbitraire.

```
let (@@) = new append;;
let double_primes = primes @@ (new nil @@ new nil) @@ primes;;

double_primes#iter print_int;;
2357111323571113- : unit = ()
```

Typage

Nous finissons cette section par une illustration du typage en Ocaml. Après avoir décrit brièvement les mécanismes principaux, nous montrerons que les constructions réputées difficiles à typer sont résolues sans difficulté en Objective ML.

L'utilité du typage est pour nous une évidence. Mais, c'est une évidence encore plus flagrande en présence d'objets, car les objets permettent de séparer des parties de programmes qui coopèrent étroitement et de surcroît récursivement.

En Objective ML, le typage des objets s'appuie très fortement et à tous les niveaux sur le polymorphisme des variables de rangée, comme pour le langage ML-ART. Le typage des messages n'y fait pas exception. Les diverses contraintes liées à l'envoi de message sont accumulées dans le type de l'objet comme dans un type enregistrement, les nouvelles contraintes s'accrochant aux précédentes en instantiant leur variable de rangée.

```
let g x = x#car;;
val g : ⟨ car : α; .. ⟩ → α = ⟨fun⟩
```

Elles introduisent à leur tour une nouvelle variable de rangée plus faible à laquelle les contraintes suivantes s'accrocheront.

```
let g x = if x#null then x#car else x#cdr;;
val g : ⟨ car : α; cdr : α; null : bool; .. ⟩ → α = ⟨fun⟩
```

Rappelons le type de la fonction `écho`, définie ci-dessus.

```
écho;;
- : ⟨ imprime : α; .. ⟩ → α = ⟨fun⟩
```

Elle peut être appliquée aux cailloux et aux bijoux, mais pas à la liste `primes`.

```
écho pierre; écho solitaire;;
Diamant Diamant Diamant 13 caratsDiamant 13 carats- : unit = ()
écho primes;;
```

Sous-typage Les langages explicitement typés utilisent souvent le sous-typage plutôt que le polymorphisme pour typer l'envoi de messages. Pour illustrer l'usage du sous-typage sur un exemple simple, il nous faut empêcher le polymorphisme artificiellement en utilisant une contrainte de type. La version monomorphe de la fonction `écho` définie ci-dessous peut-être appliquée aux cailloux mais plus aux bijoux.

```
let écho_monomorphe (x : caillou) = écho x;;
val écho_monomorphe : caillou → unit = ⟨fun⟩
écho_monomorphe pierre;;
Diamant Diamant - : unit = ()
écho_monomorphe solitaire;;
```

Dans certains langages explicitement typés le sous-typage est en partie implicite (les bornes des variables polymorphes doivent toujours être indiquées explicitement). Dans Ocaml, à l'inverse, les types sont implicites mais le sous-typage doit être explicite. Ainsi, nous devons écrire :

```
let comme_un_caillou x =  (x :> caillou);;
val comme_un_caillou :
  ⟨ concasse : char list; imprime : unit; .. ⟩ → caillou = ⟨fun⟩
```

La fonction `écho_monomorphe` peut être indirectement appliquée à des bijoux à condition de les considérer explicitement comme des cailloux, en utilisant la fonction de coercion ci-dessus ou une annotation de type :

```
écho_monomorphe (comme_un_caillou solitaire);;
Diamant 13 caratsDiamant 13 carats− : unit = ()
écho_monomorphe (solitaire :> caillou);;
Diamant 13 caratsDiamant 13 carats− : unit = ()
```

Il est possible de rentre le polymorphisme et le sous-typage simultanément implicites en utilisant un formalisme de typage avec contraintes [98], mais il reste à comprendre comment adapter le mécanisme d'abréviation automatique des types, indispensable à leur lisibilité.

La récursion est souvent difficile à typer correctement. En fait, puisque Ocaml utilise le polymorphisme paramétrique plutôt que le sous-typage, toutes les difficultés s'estompent. Nous résumons toutes les opérations autour de la récursion dans une classe démoniaque.

```
class démon =
  object (lui_même)
    val mutable gènes = Random.int 9999999
    method identité = gènes
    method même = lui_même
    method clone = {⟨ ⟩}
    method reproduction = {⟨ gènes = gènes + 1 ⟩}
    method mutation = gènes ← gènes + Random.int 9999999
  end;;
class démon :
  object (α)
    val mutable gènes : int
    method clone : α
    method identité : int
    method mutation : unit
    method même : α
    method reproduction : α
  end
```

Une instance du démon, appelons-la dolly, mémorise son identité dans ses gènes. Elle possède une méthode `même` qui retourne dolly elle-même, une méthode `clone` qui duplique dolly, et deux méthodes `reproduction` et `mutation` qui retournent une copie modifiée de dolly ou modifie dolly elle-même. Les types et les propriétés annoncées sont vérifiées ci-dessous :

```
let dolly = new démon;;
val dolly : démon = ⟨obj⟩
```

```
dolly = dolly # même && dolly ≠ dolly # clone;;
- : bool = true
let id = dolly # identité in dolly # reproduction; id = dolly # identité;;
- : bool = true
let id = dolly # identité in dolly # mutation; id ≠ dolly # identité;;
- : bool = true
```

Le démon n'est pas une abstraction de l'esprit. Chaque classe concrète peut cacher un petit démon en son sein. Pour illustration, voici une liste démoniaque.

```
module D = struct
  class [α] cons h t = object inherit démon inherit [α] L.cons h t end
  class [α] nil = object inherit démon inherit [α] L.nil end
end;;
```

suivie d'une expérimentation...

```
let expérience = new D.cons solitaire (new D.nil);;
val expérience : bijou D.cons = ⟨obj⟩
expérience # même # reproduction # clone # même # car # concasse;;
- : char list = ['D'; 'i'; 'a'; 'm'; 'a'; 'n'; 't'; ' ']
```

Les méthodes binaires s'écrivent aussi naturellement en Ocaml. Un exemple typique de méthode binaire (ici nous utilisons la comparaison polymorphe $\geq$) :

```
class compare = object (self) method better x = x ≥ self end;;
class compare : object (α) method better : α → bool end
```

peut être ajouté à la plupart des objets. Par exemple,

```
class joaillier c v = object inherit bijou c v inherit compare end;;
```

Une solutions simple a un problème réputé difficile

La simplicité d'Objective Caml n'entame en rien son expressivité. Au contraire, en s'appuyant sur le polymorphisme paramétrique et la synthétise les types, nous obtenons la garantie d'un meilleur compromis entre expressivité et simplicité. L'exemple qui suit, connu comme l'exemple du sujet et de l'observateur, a été proposé dans la littérature comme un problème difficile d'héritage car mettant en jeu des classes inter-connectées, et il a fait couler beaucoup d'encre dans la communauté Java.

La classe `observateur` possède une méthode distinguée `signale` qui attend deux arguments, un sujet et un événement pour exécuter une action.

```
class virtual ['sujet, 'événement] observateur =
  object method virtual signale : 'sujet → 'événement → unit end;;
```

Un objet de la classe `sujet` maintient à jour une liste de ses observateurs et possède une méthode `signale_observateurs` qui distribue le message `signale` à tous les observateurs avec un événement particulier `e`.

```
class ['observateur, 'événement] sujet =
  object (self)
    val mutable observateurs = ([]:'observateur list)
```

```
    method ajoute_observateur obs = observateurs ← (obs :: observateurs)
    method signale_observateurs (e : 'événement) =
        List.iter (fun x → x#signale self e) observateurs
  end;;
```

La difficulté réside habituellement dans la définition d'une instance de ce motif récursif par héritage. En objective Caml, cela se fait naturellement.

```
type event = Raise | Resize | Move;;
let string_of_event =
  function Raise → "Raise" | Resize → "Resize" | Move → "Move";;
let count = ref 0;;
class ['observateur] fenêtre_sujet =
  let id = count := succ !count; !count in
  object (self)
    inherit ['observateur, event] sujet
    val mutable position = 0
    method identité = id
    method déplace x = position ← position + x; self#signale_observateurs Move
    method dessine = Printf.printf "{Position = %d}\n"  position;
  end;;
class ['sujet] fenêtre_observateur =
  object
    inherit ['sujet, event] observateur
    method signale s e = s#dessine
  end;;
```

Les objets de la classe `fenêtre_sujet` sont sans surprise récursifs.

```
let fenêtre = new fenêtre_sujet;;
val fenêtre : ⟨ signale : α → event → unit; _.. ⟩ fenêtre_sujet as α =
  ⟨obj⟩
```

Cependant les classes `fenêtre_sujet` et `fenêtre_observateur` ne sont elles-mêmes pas récursives.

```
let fenêtre_observateur = new fenêtre_observateur;;
fenêtre#ajoute_observateur fenêtre_observateur;;
fenêtre#déplace 1;;
{Position = 1}
- : unit = ()
```

Les classes `sujet` et `fenêtre_observateur` peuvent à nouveau être étendues par héritage. Par exemple, il est possible d'ajouter un nouveau comportement au sujet et de raffiner celui de l'observateur.

```
class ['observateur] grande_fenêtre_sujet =
  object (self)
    inherit ['observateur] fenêtre_sujet
    val mutable size = 1
    method resize x = size ← size + x; self#signale_observateurs Resize
    val mutable top = false
    method raise = top ← true; self#signale_observateurs Raise
```

```
    method draw = Printf.printf "{Position = %d; Size = %d}\n"  position size;
  end;;
class ['sujet] grande_fenêtre_observateur =
  object
    inherit ['sujet] fenêtre_observateur as super
    method signale s e = if e ≠ Raise then s#raise; super#signale s e
  end;;
```

Nous pouvons aussi ajouter un nouvel observateur :

```
class ['sujet] trace_observateur =
  object
    inherit ['sujet, event] observateur
    method signale s e =
      Printf.printf "⟨Window %d ≤= %s⟩\n" s#identité (string_of_event e)
  end;;
```

et combiner l'ensemble ainsi :

```
let fenêtre = new grande_fenêtre_sujet;;
fenêtre#ajoute_observateur (new grande_fenêtre_observateur);;
fenêtre#ajoute_observateur (new trace_observateur);;
fenêtre#déplace 1; fenêtre#agrandit 2;;
```

Dans cette partie, nous avons montré l'expressivité et la souplesse d'Ocaml. Mais quelqu'en soit la réussite, il y a toujours des aspects à améliorer. Nous connaissons trois problèmes : l'absence de méthodes polymorphes, la difficulté à cacher les méthodes a posteriori, et le choix difficile entre les styles de programmation objets et modulaire.

1.6 Polymorphisme de première classe

Une limitation importante d'Ocaml qui est résolue dans le chapitre 7 est l'absence de méthodes polymorphes. Nous avons montré comment équiper les listes avec une méthode d'itération. Malheureusement, l'ajout d'une méthode `fold` n'est pas possible en Ocaml, car cela nécessite des méthodes polymorphes. Passant d'un style traditionnel à un style objet, les fonctionnelles se retrouvent être des méthodes dans un objet et ne peuvent plus être polymorphe.

Nous avons montré dans le langage ML-ART comment ajouter du polymorphisme de première classe en utilisant des constructeurs de types. Ajouter cette construction au langage Ocaml permettrait de définir des méthodes polymorphes. Cependant leur usage serait très pénible car chaque méthode polymorphe devrait être précédée d'une définition de type, et chaque utilisation d'un message polymorphe devrait mentionner explicitement le constructeur réalisant la coercion du type polymorphe vers un type ML.

Dans le chapitre 7, nous proposons une autre approche permettant d'ajouter des types d'ordre supérieur à ML, à la fois plus expressive et plus souple que celle introduite dans le chapitre 5. Bien que cette extension soit motivée par son application au langage Objective ML, elle est de façon plus générale, une extension de ML avec du polymorphisme d'ordre supérieur. Nous l'étudions en détail dans le cadre du langage ML, puis nous l'appliquons au cas des méthodes polymorphes.

Nous donnons ici une présentation plus intuitive, mais moins formelle que dans le chapitre 7. Un problèmes lié à la déclaration préalable des types polymorphes est de toujours distinguer des types polymorphes construits de façon différente même s'ils sont équivalents. Par exemple, la déclaration

$$\texttt{type } \alpha \texttt{ général} = \texttt{Général of All } \beta.\ \alpha * (\beta \to \beta)$$

définit un constructeur `Général` qui permet de coercer le type polymorphe $\texttt{All } \beta.\ \alpha * (\beta \to \beta)$ vers le type ML α `général` et inversement en utilisant le filtrage. De façon analogue, on peut définir

$$\texttt{type } \alpha \texttt{ flèche} = \texttt{Flèche of All } \beta.\ (\alpha \to \alpha) * (\beta \to \beta)$$

Cependant, les types $(\gamma \to \gamma)$ `général` et γ `flèche` sont incompatibles bien que représentant le même type polymorphe $\texttt{All } \beta.\ (\gamma \to \gamma) * (\beta \to \beta)$. Ce problème est corrigé en fournissant une notation $[\sigma]$ pour manipuler directement un type polymorphe σ comme un type ML de première classe.

$$\tau ::= \alpha \mid \tau \to \tau \mid [\sigma] \qquad\qquad \sigma ::= \tau \mid \forall\alpha.\sigma$$

Nous étendons les expressions avec les constructions suivantes

$$a ::= \ldots \mid [a : \sigma] \mid \langle a \rangle \mid (a : \tau)$$

L'expression $[a : \sigma]$ introduit une expression a de type polymorphe σ. À l'inverse, $\langle a \rangle$ coerce une expression polymorphe a en une expression ordinaire. Enfin $(a : \tau)$ force l'expression a à être de type τ.

Le principe de notre proposition est de mélanger polymorphisme explicite et polymorphisme implicite, mais sans jamais avoir à deviner le polymorphisme. En effet, cela nous entraînerait dans les problèmes bien connus d'unification d'ordre supérieur. "Deviner" signifie ici introduire un type polymorphe. Par exemple, $\texttt{fun } (x)\ \langle x^1 \rangle\ x$ n'est pas typable (les exposants sont des marques permettant d'identifier les différentes occurrences d'une même variable) parce qu'il faut deviner le type polymorphe de la variable x^1, Plus précisément, l'expression $\langle x^1 \rangle$ est rejetée parce que x^1 à un type polymorphe inconnu. Par contre, nous nous autorisons à propager le polymorphisme. L'expression

$$\texttt{let } x = [\texttt{fun } (y)\ y : \forall\alpha.\alpha \to \alpha] \texttt{ in } \langle x \rangle\ x$$

est bien typée parce que x a le type polymorphe connu $\forall\alpha.\alpha \to \alpha$. En fait, le polymorphisme ne peut être extrait par la construction $\langle \_ \rangle$ que lorsqu'il est connu. Au besoin, nous pouvons insérer des contraintes de type explicites pour indiquer (faire connaître) le polymorphisme d'une expression. Par exemple, $\texttt{fun } (x)\ \langle x : \forall\alpha.\alpha \to \alpha \rangle\ x$ est bien typé.

La difficulté consiste à formaliser les notions de « polymorphisme connu » et de « propagation ». En ML, la propagation d'information de type repose sur l'unification, bidirectionelle, verticale et transversale, alors que la notion de propagation ci-dessus est unidirectionnelle et seulement verticale. Par exemple, considérons l'expression $\texttt{fun } (x)\ \langle x^1 \rangle\ (x^2 : \forall\alpha \to \alpha)$. L'occurrence de x^2 porte une contrainte qui peut remonter vers le lieur et redescendre vers l'occurrence x^1, produisant l'effet d'une propagation transversale de x^2 vers x^1. Avec ce scénario, l'expression $\langle x \rangle$ serait typable. Cependant pour un autre scénario dans lequel la variable x^1 serait typée en premier, donc sans que la contrainte portant sur x^2 ne puisse être transmise, l'expression $\langle x \rangle$ devrait être rejetée car x a un type polymorphe inconnu à ce stade. Pour éviter un mécanisme de retour en arrière pendant le typage, nous rejetons cet exemple, donc nous refusons la liaison transversale.

Le résultat important que nous obtenons est une spécification simple de la propriété de propagation obtenue en considérant les types comme des graphes plutôt que des arbres, ou ce qui revient au même en gardant trace du partage. Nous mettons ce partage en évidence aux nœuds $[\_]$ en les marquant par une variable de type. C'est-à-dire que nous remplaçons $[\sigma]$ dans la grammaire des types par $[\sigma]^\epsilon$. Pour typer l'expression x^1 dans $\texttt{fun } (x)\ \langle x^1 \rangle\ x$, nous pouvons typer $x^1 : [\sigma]^\epsilon$ dans le contexte $x : [\sigma]^\epsilon$. Le fait que la marque ϵ de $[\sigma]$ apparaisse dans le contexte signifie que le nœud est partagé et nous refusons d'en extraire le polymorphisme. Inversement, l'expression

`let` $x = [$`fun` $(y)\ y : \forall\alpha.\alpha \to \alpha]$ `in` $\langle x^1 \rangle$ est bien typée car le type de x, $[\forall\alpha.\alpha \to \alpha]^{\epsilon'}$ est polymorphe, et la variable x^1 a le type $[\forall\alpha.\alpha \to \alpha]^{\epsilon''}$ dans l'environnement $x : [\forall\alpha.\alpha \to \alpha]^{\epsilon'}$, pour une variable ϵ'' différente de ϵ'. L'expression `fun` $(x)\ \langle x : \forall\alpha.\alpha \to \alpha \rangle\ x$ est typable à condition que la contrainte de type explicite $(\_ : \forall\alpha.\alpha \to \alpha)$ rompe le partage entre le type de l'argument et le type du résultat. Une telle contrainte se comporte comme une fonction de type $\forall\epsilon\epsilon'.[\forall\alpha.\alpha \to \alpha]^{\epsilon} \to [\forall\alpha.\alpha \to \alpha]^{\epsilon'}$.

Nous donnons une formalisation précise du système de typage dans le chapitre 7 et nous montrons qu'il admet des types principaux.

Le polymorphisme semi-explicite permet avant tout d'étendre Objective ML avec des méthodes polymorphes. En fait, cette proposition s'intègre parfaitement dans le langage qui fait déjà bon usage du partage par le mécanisme d'abréviation des types. Avec un peu de sucre syntaxique et quelques variations, on pourra y définir la classe suivante.

```
module P = struct
  class [α] cons h t = object (self) inherit [α] L.cons h t
    method (fold : All β. (α → β → β) → β → β) f x =
        self#cdr#fold f (f self#car x)
  end

  class [α] nil = object inherit [α] L.nil
    method (fold : All β. (α → β → β) → β → β) f x = x
  end
end
```

L'envoi d'un message à un objet se fait toujours par la même construction, mais il sera typé de façon polymorphe ou monomorphe selon que le type est connu et polymorphe ou, au contraire, inconnu ou monomorphe.

```
let deux = new P.cons 1 (new P.cons 2 (new nil));;
deux # fold (fun x y → new cons (x#car * x#car) y) # iter print_int;;
```

De façon plus générale, le polymorphisme semi-explicite s'applique à toutes les situations où le polymorphisme de ML est trop limité.

Dans la proposition présente, deux schémas de types qui ne sont pas égaux (au regard de leur squelettes polymorphes, les parties monomorphes pouvant être instanciées librement) sont incompatibles. En particulier, il n'est pas possible d'intancier une schéma de type par un autre qui serait moins précis. Il serait intéressant d'étendre notre proposition avec un mécanisme d'affaiblissement permettant dans certaines conditions d'unifier des types actuellement considérés comme incompatibles. C'est une direction de recherche que nous poursuivons.

1.7 Des classes aux objets par la relation de sous-typage

Objets et classes se ressemblent évidemment. De façon intuitive, on peut considérer une classe comme une fonction qui retourne un objet. Cependant, cette vision trop simplifiée des objets exclue l'héritage. Dans les langages ML-ART ou objective ML, nous avons donné aux classes des formes plus compliquées.

Une notion importante qui n'est pas prise en compte par le modèle précédent est l'abstraction. Les variables d'instance sont visibles dans les classes. Elles sont cachées automatiquement dans les objets par un mécanisme d'abstraction. Elle peuvent également être cachées dans les classes, mais explicitement.

Par exemple, on peut protéger la structure des cellules en cachant leur représentation interne par une contrainte de signature. En revanche, il est plus difficile de cacher les méthodes. Le sous-typage permet de les cacher dans les objets à condition qu'il ne reste pas d'autres méthodes binaires. Cela peut aussi être fait automatiquement, par exemple en déclarant une méthode *protégée*. Une méthode protégée se comporte comme une variable d'instance, implicitement cachée dans les objets, et pouvant être explicitement cachée dans sa classe par une contrainte de signature. Inversement, une méthode qui n'a pas été déclarée protégée ne peut pas le devenir. En effet, une autre méthode m' aurait pu supposer la méthode m visible de type τ, et l'oubli de la méthode m serait dangereux parce qu'il permettrait de redéfinir m avec un autre type τ' incompatible, brisant ainsi l'hypothèse faite par m' sur m.

Le mécanisme d'abstraction n'est pas simple. Bien que traditionnellement il se limite à l'instantiation d'une classe en un objet, on le retrouve à la fois dans les classes et les objets, avec des propriétés très similaires. Le mécanisme d'abstraction de type est compliqué par un mécanisme simultané d'abstraction de valeurs, puisque les classes sont en plus abstraites (au sens fonctionnel) par rapport à des paramètres d'initialisation. Les classes et les objets partagent des mécanismes similaires pas toujours bien dissociés. Ces mécanismes sont au centre de la notion d'abstraction actuellement trop limitée. Aussi, nous avons cherché à établir un lien plus étroit entre les objets et les classes.

Nous menons cette étude dans le chapitre 8 en considérant un calcul d'objets primitifs similaire à celui de M. Abadi et L. Cardelli [3]. Nous choisissons un système de type explicite pour nous libérer des contraintes liées à la synthèse des types. Nous montrons qu'un enrichissement de la structure des types des objets avec toute la richesse des types-enregistrements permet de tracer à la fois les types des méthodes, leur présence, et leur usage par liaison tardive.

Nous pouvons ainsi distinguer entre une méthode présente avec le type τ (mais pas utilisée) et une méthode exigée avec le type τ. La première peut être oubliée et redéfinie, éventuellement avec un autre type. La seconde peut devenir virtuelle —sa présence est oubliée mais son type est conservé— puis être redéfinie ultérieurement avec le même type; elle peut aussi être cachée (son type n'est alors plus visible), mais elle ne pourra plus être redéfinie.

L'utilisation de méthodes virtuelles au niveau des objets nous permet de distinguer entre l'introduction d'une méthode et sa définition réelle, qui peut être retardée. On peut alors considérer les classes comme des objets avec des méthodes virtuelles. Les variables d'instance sont des champs virtuels, alloués mais non définis et l'opération de création d'une instance revient à définir ces champs avec des valeurs précises. Nous identifions ainsi classes et objets, méthodes et variables d'instance.

Les résultats que nous obtenons qui doivent être approfondis, ouvrent de nouvelles directions à la fois vers la simplification des concepts, mais aussi vers plus de flexibilité et d'expressivité dans les opérations d'abstraction.

Chapter 2

Synthèse des types enregistrements dans une extension naturelle de ML

Ce chapitre a été publié dans [106].

Synthèse des types enregistrements dans une extension naturelle de ML

Nous proposons une extension de ML avec des enregistrements où l'héritage est obtenu par le polymorphisme générique de ML. Toutes les opérations sur les enregistrements, excepté la concaténation, sont traitées sans restriction, y compris de nouvelles opérations telles que le renommage des champs. La solution proposée repose sur une généralisation des types qui sont simultanément munis de sortes et considérés modulo une théorie équationnelle et qui conduit à la notion de types-enregistrements. La solution est simple et modulaire, et l'algorithme d'inférence est efficace en pratique.

Typeckecking records in a natural extension of ML

We describe an extension of ML with records where inheritance is given by ML generic polymorphism. All common operations on records but concatenation are supported, in particular the free extension of records. Other operations such as renaming of fields are added. The solution relies on an extension of ML, where the language of types is sorted and considered modulo equations, and on a record extension of types. The solution is simple and modular and the type inference algorithm is efficient in practice.

Introduction

The aim of typechecking is to guarantee that well-typed programs will not produce runtime errors. A type error is usually due to a programmer's mistake, and thus typechecking also helps him in debugging his programs. Some programmers do not like writing the types of their programs by hand. In the ML language for instance, type inference requires as little type information as the declaration of data structures; then all types of programs will be automatically computed.

Our goal is to provide type inference for labeled products, a data structure commonly called records, allowing some inheritance between them: records with more labels should be allowed where records with fewer labels are required.

After defining the operations on records and recalling related work, we first review the solution for a finite (and small) set of labels, which was presented in [99], then we extend it to a denumerable set of labels. In the last part we discuss the power and weakness of the solution, we describe some variations, and suggest improvements.

Without records, data structures are built using product types, as in ML, for instance.

("Peter", "John", "Professor", 27, 5467567, 56478356, ("toyota", "old", 8929901))

With records one would write, instead:

{name = "Peter"; lastname = "John"; job = "Professor"; age = 27; id = 5467567;
license = 56478356; vehicle = {name = "Toyota"; id = 8929901; age = "old"}}

The latter program is definitely more readable than the former. It is also more precise, since components are named. Records can also be used to name several arguments or several results of a function. More generally, in communication between processes records permit the naming of the different ports on which processes can exchange information. One nice example of this is the LCS language [7], which is a combination of ML and Milner's CCS [78].

Besides typechecking records, the challenge is to avoid record type declarations and fix size records. Extensible records introduced by Wand [121, 27] can be built from older records by adding new fields. This feature is the basis of inheritance in the view of objects as records [121, 27].

The main operations on records are introduced by examples, using a syntax similar to CAML syntax [33, 122]. Like variable names, labels do not have particular meanings, though choosing good names (good is subjective) helps in writing and reading programs. Names can, of course, be reused in different records, even to build fields of different types. This is illustrated in the following three examples:

let car = {name = "Toyota"; age = "old"; id = 7866};;

let truck = {name = "Blazer"; id = 6587867567};;

let person = {name = "Tim"; age = 31; id = 5656787};;

Remark that no declaration is required before the use of labels. The record person is defined on exactly the same fields as the record car, though those fields do not have the same intuitive meaning. The field age holds values of different types in car and in person.

All these records have been created in one step. Records can also be build from older ones. For instance, a value driver can be defined as being a copy of the record person but with one more field, vehicle, filled with the previously defined car object.

let driver = {person with vehicle = car};;

Note that there is no sharing between the records person and driver. You can simply think as if the former were copied into a new empty record before adding a field car to build the latter. This

construction is called the *extension* of a record with a new field. In this example the newly defined field was not present in the record person, but that should not be a restriction. For instance, if our driver needs a more robust vehicle, we write:

```
let truck_driver = {driver with vehicle = truck};;
```

As previously, the operation is not a physical replacement of the vehicle field by a new value. We do not wish the old and the new value of the vehicle field to have the same type. To distinguish between the two kinds of extensions of a record with a new field, we will say that the extension is *strict* when the new field must not be previously defined, and *free* otherwise.

A more general operation than extension is concatenation, which constructs a new record from two previously defined ones, taking the union of their defined fields. If the car has a rusty body but a good engine, one could think of building the hybrid vehicle:

```
let repaired_truck = {car and truck};;
```

This raises the question: what value should be assigned to fields which are defined in both car and truck? When there is a conflict (the same field is defined in both records), priority could be given to the last record. As with free extension, the last record would eventually overwrite fields of the first one. But one might also expect a typechecker to prevent this situation from happening. Although concatenation is less common in the literature, probably because it causes more trouble, it seems interesting in some cases. Concatenation is used in the standard ML language [48] when a structure is opened and extended with another one. In the LCS language, the visible ports of two processes run in parallel are exactly the ports visible in any of them. And as shown by Mitchell Wand [121] multiple inheritance can be coded with concatenation.

The constructions described above are not exhaustive but are the most common ones. We should also mention the permutation, renaming and erasure of fields. We described how to build records, but of course we also want to read them. There is actually a unique construction for this purpose.

```
let id x = x.id;;                    let age x = x.age;;
```

Accessing some field a of a record x can be abstracted over x, but not over a: Labels are not values and there is no function which could take a label as argument and would access the field of some fixed record corresponding to that label. Thus, we need one extraction function per label, as for id and age above. Then, they can be applied to different records of different types but all possessing the field to access. For instance,

```
age person, age driver;;
```

They can also be passed to other functions, as in:

```
let car_info field = field car;;     car_info age;;
```

The testing function eq below should of course accept arguments of different types provided they have an id field of the same type.

```
let eq x y = equal x.id y.id;;       eq car truck;;
```

These examples were very simple. We will typecheck them below, but we will also meet more tricky ones.

Related work

Luca Cardelli has always claimed that functional languages should have record operations. In 1986, when he designed Amber, his choice was to provide the language with records rather than polymorphism. Later, he introduced bounded quantification in the language FUN , which he extended to higher order bounded quantification in the language QUEST. Bounded quantification is an extension of ordinary quantification where quantified variables range in the subset of types that are all subtypes of the bound. The subtyping relation is a lattice on types. In this language, subtyping is essential for having some inheritance between records. A slight but significant improvement of bounded quantification has been made in [19] to better consider recursive objects; a more general but less tractable system was studied by Pavel Curtis [34]. Today, the trend seems to be the simplification rather than the enrichment of existing systems [26, 49, 25]. For instance, an interesting goal was to remove the subtype relation in bounded quantification [49]. Records have also been formulated with explicit labeled conjunctive types in the language Forsythe [116].

In contrast, records in implicitly typed languages have been less studied, and the proposed extensions of ML are still very restrictive. The language Amber [20, 21] is monomorphic and inheritance is obtained by type inclusion. A major step toward combining records and type inference has been Wand's proposal [119] where inheritance is obtained from ML generic polymorphism. Though type inference is incomplete for this system, it remains a reference, for it was the first concrete proposal for extending ML with records having inheritance. The year after, complete type inference algorithms were found for a strong restriction of this system [56, 86]. The restriction only allows the strict extension of a record. Then, the author proposed a complete type inference algorithm for Wand's system [99], but it was formalized only in the case of a finite set of labels (a previous solution given by Wand in 1988 did not admit principal types but complete sets of principal types, and was exponential in size in practice). Mitchell Wand revisited this approach and extended it with an "and" operation[1] but did not provide correctness proofs. The case of an infinite set of labels has been addressed in [100], which we review in this article.

2.1 A simple solution when the set of labels is finite

Though the solution below will be made obsolete by the extension to a denumerable set of labels, we choose to present it first, since it is very simple and the extension will be based on the same ideas. It will also be a decent solution in cases where only few labels are needed. And it will emphasize a method for getting more polymorphism in ML (in fact, we will not put more polymorphism in ML but we will make more use of it, sometimes in unexpected ways).

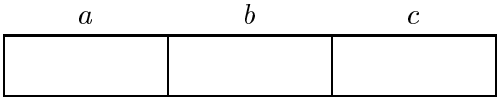
We will sketch the path from Wand's proposal to this solution, for it may be of some interest to describe the method which we think could be applied in other situations. As intuitions are rather subjective, and ours may not be yours, the section 2.1.1 can be skipped whenever it does not help.

2.1.1 The method

Records are partial functions from a set $\mathcal{L}$ of labels to the set of values. We simplify the problem by considering only three labels a, b and c. Records can be represented in three field boxes, once

[1] It can be understood as an "append" on association lists in lisp compared to the "with" operation which should be understood as a "cons".

labels have been ordered:

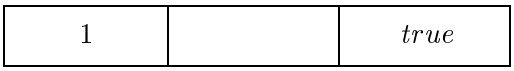


Defining a record is the same as filling some of the fields with values. For example, we will put the values 1 and *true* in the a and c fields respectively and leave the b field undefined.

Typechecking means forgetting some information about values. For instance, it does not distinguish two numbers but only remember them as being numbers. The structure of types usually reflects the structure of values, but with fewer details. It is thus natural to type record values with partial functions from labels ($\mathcal{L}$) to types ($\mathcal{T}$), that is, elements of $\mathcal{L} \rightharpoonup \mathcal{T}$. We first make record types total functions on labels using an explicitly undefined constant abs ("absent"): $\mathcal{L} \longrightarrow \mathcal{T} \cup \{abs\}$. In fact, we replace the union by the sum $pre(\mathcal{T}) + abs$. Finally, we decompose record types as follows:

$$\mathcal{L} \longrightarrow [1, Card(\mathcal{L})] \longrightarrow pre(\mathcal{T}) + abs$$

The first function is an ordering from $\mathcal{L}$ to the segment $[1, Card(\mathcal{L})]$ and can be set once and for all. Thus record types can be represented only by the second component, which is a tuple of length $Card(\mathcal{L})$ of types in $pre(\mathcal{T}) + abs$. The previous example is typed by



$$\Pi(\ pre(num)\ ,\quad abs\quad ,\ pre(bool)\ )$$

A function $\_.a$ reading the a field accepts as argument any record having the a field defined with a value M, and returns M. The a field of the type of the argument must be $pre(\tau)$ if τ is the type of M. We do not care whether other fields are defined or not, so their types may be anything. We choose to represent them by variables θ and ε. The result has type α.

$$\_.a : \Pi(pre(\alpha), \theta, \varepsilon) \to \alpha$$

2.1.2 A formulation

We are given a collection of symbols $\mathcal{C}$ with their arities $(\mathcal{C}^n)_{n \in \mathbf{N}}$ that contains at least an arrow symbol $\to$ of arity 2, a unary symbol pre and a nullary symbol abs. We are also given two sorts *type* and *field*. The signature of a symbol is a sequence of sorts, written ι for a nullary symbol and $\iota_1 \ldots \otimes \iota_n \Rightarrow \iota$ for a symbol of arity n. The signature $\mathcal{S}$ is defined by the following assertions (we write $\mathcal{S} \vdash f :: \iota$ for $(f, \iota) \in \mathcal{S}$):

$$\begin{array}{ll}
\mathcal{S} \vdash pre :: type \Rightarrow field & \\
\mathcal{S} \vdash abs :: field & \\
\mathcal{S} \vdash \Pi :: field^{\mathrm{card}(\mathcal{L})} \Rightarrow type & \\
\mathcal{S} \vdash f :: type^n \Rightarrow type & f \in \mathcal{C}^n \setminus \{pre, abs, \Pi\}
\end{array}$$

The language of types is the free sorted algebra $\mathcal{T}(\mathcal{S}, \mathcal{V})$. The extension of ML with sorted types is straightforward. We will not formalize it further, since this will be subsumed in the next section.

The inference rules are the same as in ML though the language of types is sorted. The typing relation defined by these rules is still decidable and admits principal typings (see next section for a precise formulation). In this language, we assume the following primitive environment:

$$\begin{gathered}\{\} : \Pi\,(abs, \ldots abs\,) \\ \_.a : \Pi\,(\theta_1 \ldots,\, pre\,(\alpha),\, \ldots \theta_l) \to \alpha \\ \{\_\ with\ a = \_\} : \Pi\,(\theta_1, \ldots \theta_l) \to \alpha \to \Pi\,(\theta_1 \ldots,\, pre\,(\alpha),\, \ldots \theta_l)\end{gathered}$$

Basic constants for $\Pi\mathrm{ML}_{fin}$

The constant $\{\}$ is the empty record. The $\_.a$ constant reads the a field from its argument, we write $r.a$ the application $(\_.a)\ r$. Similarly $\{r\ with\ a = M\}$ extends the records r on label a with value M.

2.2 Extension to large records

Though the previous solution is simple, and perfect when there are only two or three labels involved, it is clearly no longer acceptable when the set of labels is getting larger. This is because the size of record types is proportional to the size of this set — even for the type of the null record, which has no field defined. When a local use of records is needed, labels may be fewer than ten and the solution works perfectly. But in large systems where some records are used globally, the number of labels will quickly be over one hundred.

In any program, the number of labels will always be finite, but with modular programming, the whole set of labels is not known at the beginning (though in this case, some of the labels may be local to a module and solved independently). In practice, it is thus interesting to reason on an "open", i.e. countable, set of labels. From a theoretical point of view, it is the only way to avoid reasoning outside of the formalism and show that any computation done in a system with a small set of labels would still be valid in a system with a larger set of labels, and that the typing in the latter case could be deduced from the typing in the former case. A better solution consists in working in a system where all potential labels are taken into account from the beginning.

In the first part, we will illustrate the discussion above and describe the intuitions. Then we formalize the solution in three steps. First we extend types with record types in a more general framework of sorted algebras; record types will be sorted types modulo equations. The next step describes an extension of ML with sorts and equations on types. Last, we apply the results to a special case, re-using the same encoding as for the finite case.

2.2.1 An intuitive approach

We first assume that there are only two labels a and b. Let r be the record $\{a = 1\ ;\ b = true\}$ and f the function that reads the a field. Assuming f has type $\tau \to \tau'$ and r has type σ, f can be applied to r if the two types τ and σ are unifiable. In our example, we have

$$\begin{aligned}\tau &= \Pi\,(a : pre\,(\alpha)\ ;\ b : \theta_b)\,, \\ \sigma &= \Pi\,(a : pre\,(num)\ ;\ b : pre\,(bool))\,,\end{aligned}$$

and τ' is equal to α. The unification of τ and σ is done field by field and their most general unifier is:

$$\begin{cases}\alpha \mapsto num \\ \theta_b \mapsto pre\,(bool)\end{cases}$$

If we had one more label c, the types τ and σ would be

$$\tau = \Pi\,(a : pre\,(\alpha)\ ;\ b : \theta_b\ ;\ c : \theta_c)\,,$$
$$\sigma = \Pi\,(a : pre\,(num)\ ;\ b : pre\,(bool)\ ;\ c : abs\,)\,.$$

and their most general unifier

$$\begin{cases} \alpha \mapsto num \\ \theta_b \mapsto pre\,(bool) \\ \theta_c \mapsto abs \end{cases}$$

We can play again with one more label d. The types would be

$$\tau = \Pi\,(a : pre\,(\alpha)\ ;\ b : \theta_b\ ;\ c : \theta_c\ ;\ d : \theta_d)\,,$$
$$\sigma = \Pi\,(a : pre\,(num)\ ;\ b : pre\,(bool)\ ;\ c : abs\ \ ;\ d : abs\,)\,.$$

whose most general unifier is:

$$\begin{cases} \alpha \mapsto num \\ \theta_b \mapsto pre\,(bool) \\ \theta_c \mapsto abs \\ \theta_d \mapsto abs \end{cases}$$

Since labels c and d appear neither in the expressions r nor in f, it is clear that fields c and d behave the same, and that all their type components in the types of f and r are equal up to renaming of variables (they are isomorphic types). So we can guess the component of the most general unifier on any new field ℓ simply by taking a copy of its component on the c field or on the d field. Instead of writing types of all fields, we only need to write a template type for all fields whose types are isomorphic, in addition to the types of significant fields, that is those which are not isomorphic to the template.

$$\tau = \Pi\,(a : pre\,(\alpha)\ ;\ b : \theta_b\ ;\ \infty : \theta_\infty)\,,$$
$$\sigma = \Pi\,(a : pre\,(num)\ ;\ b : pre\,(bool)\ ;\ \infty : abs\,)\,.$$

The expression $\Pi\,((\ell : \tau_\ell)_{\ell\in I}\ ;\ \infty : \sigma_\infty)$ should be read as

$$\prod_{\ell\in\mathcal{L}} \left(\ell : \begin{cases} \tau_\ell & \text{if } \ell \in I \\ \sigma_\ell & \text{otherwise, where } \sigma_\ell \text{ is a copy of } \sigma_\infty \end{cases}\right)$$

The most general unifier can be computed without developing this expression, thus allowing the set of labels to be infinite. We summarize the successive steps studied above in this figure:

Labels	a	b	c	d	∞
τ	$pre\,(\alpha)$	θ_b	θ_c	θ_d	θ_∞
σ	$pre\,(num)$	$pre\,(bool)$	abs	abs	abs
$\tau \wedge \sigma$	$pre\,(num)$	$pre\,(bool)$	abs	abs	abs

This approach is so intuitive that it seems very simple. There is a difficulty though, due to the sharing between templates. Sometimes a field has to be extracted from its template, because it must be unified with a significant field.

The macroscopic operation that we need is the transformation of a template τ into a copy τ' (the type of the extracted field) and another copy τ'' (the new template). We regenerate the

template during an extraction mainly because of sharing. But it is also intuitive that once a field has been extracted, the retained template should remember that, and thus it cannot be the same. In order to keep sharing, we must extract a field step by step, starting from the leaves.

For a template variable α, the extraction consists in replacing that variable by two fresh variables β and γ, more precisely by the term $\ell : \beta \ ; \ \gamma$. This is exactly the substitution

$$\alpha \mapsto \ell : \beta \ ; \ \gamma$$

For a term $f(\alpha)$, assuming that we have already extracted field ℓ from α, i.e. we have $f(\ell : \beta \ ; \ \gamma)$, we now want to replace it by $\ell : f(\alpha) \ ; \ f(\gamma)$. The solution is simply to ask it to be true, that is, to assume the axiom

$$f(\ell : \beta \ ; \ \gamma) = \ell : f(\alpha) \ ; \ f(\gamma)$$

for every given symbol f but Π.

2.2.2 Extending a free algebra with a record algebra

The intuitions of previous sections are formalized by the algebra of record terms. The algebra of record terms is introduced for an arbitrary free algebra; record types are an instance. The record algebra was introduced in [100] and revisited in [104]. We summarize it below but we recommend [104] for a more thorough presentation.

We are given a set of variables $\mathcal{V}$ and a set of symbols $\mathcal{C}$ with their arities $(\mathcal{C}_n)_{n \in N}$.

Raw terms

We call *unsorted record terms* the terms of the free unsorted algebra $\mathcal{T}'(\mathcal{D}', \mathcal{V})$ where $\mathcal{D}'$ is the set of symbols composed of $\mathcal{C}$ plus a unary symbol Π and a collection of projection symbols $\{(\ell : \_ \ ; \ \_) \mid \ell \in \mathcal{L}\}$ of arity two. Projection symbols associate to the right, that is $(a : \tau \ ; \ b : \sigma \ ; \ \tau')$ stands for $(a : \tau \ ; \ (b : \sigma \ ; \ \tau'))$.

Example 1 The expressions

$$\Pi\,(a : pre\,(num) \ ; \ c : pre\,(bool) \ ; \ abs\,) \qquad \text{and} \qquad \Pi\,(a : pre\,(b : num \ ; \ num) \ ; \ abs\,)$$

are raw terms. In section 2.2.4 we will consider the former as a possible type for the record $\{a = 1 \ ; \ c = true\}$ but we will not give a meaning to the latter. There are too many raw terms. The raw term $\{a : \alpha \ ; \ \chi\} \to \chi$ must also be rejected since the template composed of the raw variable χ should define the a field on the right but should not on the left. We define record terms using sorts to constrain their formation. Only a few of the raw terms will have associated record terms.

Record terms

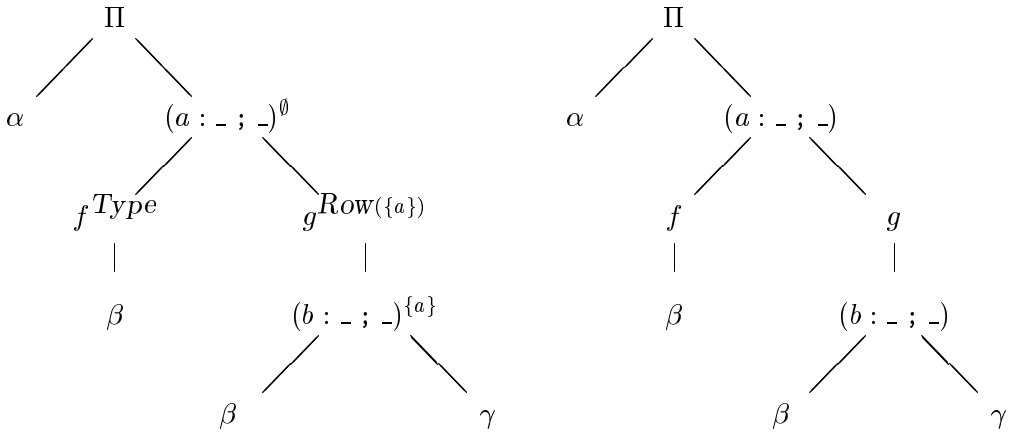
Let $\mathcal{L}$ be a denumerable set of labels. Let $\mathcal{K}$ be composed of a sort $type$, and a finite collection of sorts $(row\,(L))$ where L range over finite subsets of labels. Let $\mathcal{S}$ be the signature composed of the following symbols given with their sorts:

$$\begin{array}{ll}
\mathcal{S} \vdash \Pi :: Row(\emptyset) \Rightarrow Type & \\
\mathcal{S} \vdash f^K :: K^n \Rightarrow K & f \in \mathcal{C}^n, K \in \mathcal{K} \\
\mathcal{S} \vdash (\ell^L : _ \ ; \ _) :: Type \otimes Row(L \cup \{\ell\}) \Rightarrow Row(L) & \ell \in \mathcal{L}, L \in \mathcal{P}_{fin}(\mathcal{L} \setminus \{\ell\})
\end{array}$$

The superscripts are parts of symbols, so that the signature $\mathcal{S}$ is not overloaded, that is, every symbol has a unique signature. We write $\mathcal{D}$ the set of symbols in $\mathcal{S}$.

Definition 1 *Record terms* are the terms of the free sorted algebra $\mathcal{T}(\mathcal{S}, \mathcal{V})$. □

Example 2 The left term below is a record term. On the right, we drew a raw term with the same structure.



Script erasure

To any record term, we associate the raw term obtained by erasing all superscripts of symbols. Conversely, for any raw term τ', and any sort ι there is at most one record term whose erasure is τ'. Thus any record term τ of sort ι is completely defined by its erasure τ' and the sort ι. In the rest of the paper we will mostly use this convention. Moreover we usually drop the sort whenever it is implicit from context.

Example 3 The erasure of

$$\Pi\left(a^{\emptyset} : f^{Type}(g^{Type}) \ ; \ \left(c^{\{a\}} : f^{Type}(\alpha) \ ; \ h^{Row(\{a,c\})}\right)\right)$$

is the raw term

$$\Pi\left(a : f(g) \ ; \ c : f(\alpha) \ ; \ h\right)$$

There is no record term whose erasure would be

$$\Pi\left(a : f(b : g \ ; \ \alpha) \ ; \ h\right)$$

Record algebra

The permutation and the extraction of fields in record terms will be obtained by equations, of left commutativity and distributivity respectively. Precisely, let E be the set of axioms

- Left commutativity. For any labels a and b and any finite subset of labels L that do not contain a and b,

$$a^L : \alpha \ ; \ \left(b^{L\cup\{a\}} : \beta \ ; \ \gamma\right) = b^L : \beta \ ; \ \left(a^{L\cup\{b\}} : \alpha \ ; \ \gamma\right)$$

- Distributivity. For any symbol f, any label a and any finite subset of labels L that do not contain a,

$$f^{Row(L)}(a^L : \alpha_1 \ ; \ \beta_1, \ldots a^L : \alpha_p \ ; \ \beta_p) = a^L : f^{Type}(\alpha_1, \ldots \alpha_p) \ ; \ f^{Row(L\cup\{a\})}(\beta_1, \ldots \beta_p)$$

With the raw notation the equations are written:

- Left commutativity. At any sort $row(L)$, where L does not contain labels a and b:

$$a : \alpha \ ; \ (b : \beta \ ; \ \gamma) = b : \beta \ ; \ (a : \alpha \ ; \ \gamma)$$

- Distributivity. At any sort $row(L)$ where L does not contain label a, and for any symbol f:

$$f(a : \alpha_1 \ ; \ \beta_1, \ldots a : \alpha_p \ ; \ \beta_p) = a : f(\alpha_1, \ldots \alpha_p) \ ; \ f(\beta_1, \ldots \beta_p)$$

All axioms are regular, that is, the set of variables of both sides of equations are always identical.

Example 4 In the term

$$\Pi\,(a : pre(num) \ ; \ c : pre(bool) \ ; \ abs\,)$$

we can replace abs by $b : abs \ ; \ abs$ using distributivity, and use left commutativity to end with the term:

$$\Pi\,(a : pre(num) \ ; \ b : abs \ ; \ c : pre(bool) \ ; \ abs\,)$$

In the term

$$\Pi\,(a : pre(\alpha) \ ; \ \theta)$$

we can substitute θ by $b : \theta_b \ ; \ c \ ; \ \theta_c \ ; \ \varepsilon$ to get

$$\Pi\,(a : pre(\alpha) \ ; \ b : \theta_b \ ; \ c : \theta_c \ ; \ \varepsilon)$$

which can then be unified with the previous term field by field.

Definition 2 The algebra of record terms is the algebra $\mathcal{T}(\mathcal{S}, \mathcal{V})$ modulo the equational theory E, written $\mathcal{T}(\mathcal{S}, \mathcal{V})/E$. □

Unification in the algebra of record terms has been studied in [104].

Theorem 4 *Unification in the record algebra is decidable and unitary (every solvable unification problem has a principal unifier).*

A unification algorithm is given in the appendix.

Instances of record terms

The construction of the record algebra is parameterized by the initial set of symbols $\mathcal{C}$, from which the signature $\mathcal{S}$ is deduced. The signature $\mathcal{S}$ may also be restricted by a signature $\mathcal{S}'$ that is compatible with the equations E, that is, a signature $\mathcal{S}'$ such that for all axioms r and all sorts ι of $\mathcal{S}'$,

$$\mathcal{S}' \vdash r^l :: \iota \iff \mathcal{S}' \vdash r^r :: \iota$$

The algebra $(\mathcal{T}/E)\restriction\mathcal{S}'$ and $(\mathcal{T}\restriction\mathcal{S}')/(E\restriction\mathcal{S}')$ are then isomorphic, and consequently unification in $(\mathcal{T}\restriction\mathcal{S}')/(E\restriction\mathcal{S}')$ is decidable and unitary, and solved by the same algorithm as in $\mathcal{T}/E$. The $\mathcal{S}'$-record algebra is the restriction $\mathcal{T}(\mathcal{S}, \mathcal{V})\restriction\mathcal{S}'$ of the record algebra by a compatible signature $\mathcal{S}'$.

We now consider a particular instance of record algebra, where fields are distinguished from arbitrary types, and structured as in section 2.1. The signature $\mathcal{S}'$ distinguishes a constant symbol abs and a unary symbol pre in $\mathcal{C}$, and is defined with two sorts $type$ and $field$:

$$\begin{array}{ll}
\mathcal{S}' \vdash \Pi :: field \Rightarrow type & \\
\mathcal{S}' \vdash abs^{\iota} :: field & \iota \in \mathcal{K} \\
\mathcal{S}' \vdash pre :: type \Rightarrow field & \\
\mathcal{S}' \vdash f^{Type} :: type^{n} \Rightarrow type & f \in \mathcal{C}^n \setminus \{abs, pre, \Pi\} \\
\mathcal{S}' \vdash (\ell^{L} : _ \ ; \ _) :: field \otimes field \Rightarrow field & \ell \in \mathcal{L}, L \in \mathcal{P}_{fin}(\mathcal{L} \setminus \{\ell\})
\end{array}$$

The signature $\mathcal{S}'$ is compatible with the equations of the record algebra. We call *record types* the $\mathcal{S}'$-record algebra.

In fact, record types have a very simple structure. Terms of the sort $Row(L)$ are either of depth 0 (reduced to a variable or a symbol) or are of the form $(a : \tau \ ; \ \tau')$. By induction, they are always of the form

$$(a_1 : \tau_1 \ ; \ \ldots a_p : \tau_p \ ; \ \sigma)$$

where σ is either abs or a variable, including the case where p is zero and the term is reduced to σ. Record types are also generated by the pseudo-BNF grammar:

$$\begin{array}{lll}
\tau ::= \alpha \mid \tau \to \tau \mid \Pi\rho^{\emptyset} & & \text{types} \\
\rho^{L} ::= \chi^{L} \mid abs^{L} \mid a : \varphi \ ; \ \rho^{L \cup \{a\}} & a \notin L & \text{rows} \\
\varphi ::= \theta \mid abs \mid pre(\tau) & & \text{fields}
\end{array}$$

where α, β, γ and δ are type variables, χ, π and ξ are row variables and θ and ε are field variables. We prefer the algebraic approach which is more general.

2.2.3 Extending the types of ML with a sorted equational theory

In this section we consider a sorted regular theory $\mathcal{T}/E$ for which unification is decidable and unitary. A regular theory is one whose left and right hand sides of axioms always have the same set of variables. For any term τ of $\mathcal{T}/E$ we write $\mathcal{V}(\tau)$ for the set of its variables. We privilege a sort $Type$.

The addition of a sorted equational theory to the types of ML has been studied in [100, 102]. We recall here the main definitions and results. The language ML that we study is lambda-calculus extended with constants and a LET construct in order to mark some of the redexes, namely:

$M ::=$	Terms	M, N
$\quad x$	Variable	x, y
$\quad \mid c$	Constant	c
$\quad \mid \lambda x.\ M$	Abstraction	
$\quad \mid M\ M$	Application	
$\quad \mid let\ x = M\ in\ M$	Let binding	

The letter W ranges over finite set of variables. Type schemes are pairs noted $\forall W \cdot \tau$ of a set of variables and a term τ. The symbol $\forall$ is treated as a binder and we consider type schemes equal modulo α-conversion. The sort of a type scheme $\forall W \cdot \tau$ is the sort of τ. Contexts as sequences of assertions, that is, pairs of a term variable and a type. We write $\mathcal{A}$ the set of contexts.

Every constant c comes with a closed type scheme $\forall W \cdot \tau$, written $c : \forall W \cdot \tau$. We write B the collection of all such constant assertions. We define a relation $\vdash$ on $\mathcal{A} \times \mathrm{ML} \times \mathcal{T}$ and parameterized by B as the smallest relation that satisfies the following rules:

$$\frac{x : \forall W \cdot \tau \in A \qquad \mu : W \to \mathcal{T}}{A \vdash_S x : \mu(\tau)} \text{ (VAR-INST)} \qquad \frac{c : \forall W \cdot \tau \in B \qquad \mu : W \to \mathcal{T}}{A \vdash_S c : \mu(\tau)} \text{ (CONST-INST)}$$

$$\frac{A[x : \tau] \vdash M : \sigma \qquad \tau \in \mathcal{T}}{A \vdash \lambda x.\, M : \tau \to \sigma} \text{ (FUN)} \qquad \frac{A \vdash M : \sigma \to \tau \qquad A \vdash N : \sigma}{A \vdash M\ N : \tau} \text{ (APP)}$$

$$\frac{A \vdash_S M : \tau \qquad A[x : \forall W \cdot \tau] \vdash_S N : \sigma \qquad W \cap \mathcal{V}(A) = \emptyset}{A \vdash_S \mathit{let}\ x = M\ \mathit{in}\ N : \sigma} \text{ (LET-GEN)}$$

$$\frac{A \vdash M : \sigma \qquad \sigma =_E \tau}{A \vdash M : \tau} \text{ (EQUAL)}$$

They are the usual rules for ML except the rule EQUAL that is added since the equality on types is taken modulo the equations E.

A typing problem is a triple of $\mathcal{A} \times \mathrm{ML} \times \mathcal{T}$ written $A \triangleright M : \tau$. The application of a substitution μ to a typing problem $A \triangleright M : \tau$ is the typing problem $\mu(A) \triangleright M : \mu(\tau)$, where substitution of a context is understood pointwise and only affects the type part of assertions. A solution of a typing problem $A \triangleright M : \tau$ is a substitution μ such that $\mu(A) \vdash M : \mu(\tau)$. It is principal if all other solutions are obtained by left composition with μ of an arbitrary solution.

Theorem 5 (principal typings) *If the sorted theory $\mathcal{T}/E$ is regular and its unification is decidable and unitary, then the relation $\vdash$ admits principal typings, that is, any solvable typing problem has a principal solution.*

Moreover, there is an algorithm that given a typing problem computes a principal solution if one exists, or returns failure otherwise.

An algorithm can be obtained by replacing free unification by unification in the algebra of record terms in the core-ML type inference algorithm. A clever algorithm for type inference is described in [104].

2.2.4 Typechecking record operations

Using the two preceding results, we extend the types of ML with record types assuming given the following basic constants:

$$\begin{gathered} \{\} : \Pi\,(abs\,) \\ \_.a : \Pi\,(a : pre\,(\alpha)\ ;\ \theta) \to \alpha \\ \{\_\ \mathit{with}\ a = \_\} : \Pi\,(a : \theta\ ;\ \chi) \to \alpha \to \Pi\,(a : pre\,(\alpha)\ ;\ \chi) \end{gathered}$$

Basic constants for ΠML

There are countably many constants. We write $\{a_1 = x_1\ ;\ \ldots a_n = x_n\}$ as syntactic sugar for:

$$\{\{a_1 = x_1\ ;\ \ldots a_{n-1} = x_{n-1}\}\ \mathit{with}\ a_n : x_n\}$$

We illustrate this system by examples in the next section.

The equational theory of record types is regular, and has a decidable and unitary unification. It follows from theorems 5 and 4 that the typing relation of this language admits principal typings, and has a decidable type inference algorithm.

2.3 Programming with records

We first show on simple examples how most of the constructions described in the introduction are typed, then we meet the limitations of this system. Some of them can be cured by slightly improving the encoding. Finally, we propose and discuss some further extensions.

2.3.1 Typing examples

A typechecking prototype has been implemented in the CAML language. It was used to automatically type all the examples presented here and preceded by the # character. In programs, type variables are printed according to their sort in $\mathcal{S}'$. Letters χ, π and ξ are used for field variables and letters α, β, etc. are used for variables of the sort *type*. We start with simple examples and end with a short program.

Simple record values can be built as follows:

```
#let car = {name = "Toyota"; age = "old"; id = 7866};;
car : Pi (name : pre (string); id : pre (num); age : pre (string); abs)

#let truck = {name = "Blazer"; id = 6587867567};;
truck : Pi (name : pre (string); id : pre (num); abs)

#let person = {name = "Tim"; age = 31; id = 5656787};;
person : Pi (name : pre (string); id : pre (num); age : pre (num); abs)
```

Each field defined with a value of type τ is significant and typed with $pre(\tau)$. Other fields are insignificant, and their types are gathered in the template *abs*. The record person can be extended with a new field vehicle:

```
#let driver = {person with vehicle = car};;
driver :
  Pi (vehicle : pre (Pi (name : pre (string); id : pre (num); age : pre (string); abs));
      name : pre (string); id : pre (num); age : pre (num); abs)
```

This is possible whether this field was previously undefined as above, or defined as in:

```
#let truck_driver = {driver with vehicle = truck};;
truck_driver :
  Pi (vehicle : pre (Pi (name : pre (string); id : pre (num); abs)); name : pre (string);
      id : pre (num); age : pre (num); abs)
```

The concatenation of two records is not provided by this system.

The sole construction for accessing fields is the "dot" operation.

```
#let age x = x.age;;
age : Pi (age : pre (α); χ) → α

#let id x = x.id;;
id : Pi (id : pre (α); χ) → α
```

The accessed field must be defined with a value of type α, so it has type pre (α), and other fields may or may not be defined; they are described by a template variable χ. The returned value has type α. As any value, age can be sent as an argument to another function:

```
#let car_info field = field car;;
car_info : (Pi (name : pre (string); id : pre (num); age : pre (string); abs) → α) → α

#car_info age;;
it : string
```

The function equal below takes two records both possessing an id field of the same type, and possibly other fields. For simplicity of examples we assume given a polymorphic equality equal.

```
#let eq x y = equal x.id y.id;;
eq : Pi (id : pre (α); χ) → Pi (id : pre (α); π) → bool

#eq car truck;;
it : bool
```

We will show more examples in section 2.3.3.

2.3.2 Limitations

There are two sorts of limitations, one is due to the encoding method, the other one results from ML generic polymorphism. The only source of polymorphism in record operations is generic polymorphism. A field defined with a value of type τ in a record object is typed by $pre(\tau)$. Thus, once a field has been defined every function must see it defined. This forbids merging two records with different sets of defined fields. We will use the following function to shorten examples:

```
#let choice x y = if true then x else y;;
choice : α → α → α
```

Typechecking fails with:

```
#choice car truck;;
Typechecking error: collision between pre (string) and abs
```

The age field is undefined in truck but defined in car. This is really a weakness, since the program

```
#(choice car truck).name;;
Typechecking error: collision between pre (string) and abs
```

which should be equivalent to the program

```
#choice car.name truck.name;;
it : string
```

may actually be useful. We will partially solve this problem in section 2.3.3. A natural generalization of the eq function defined above is to abstract over the field that is used for testing equality

```
#let field_eq field x y = equal (field x) (field y);;
field_eq : (α → β) → α → α → bool
```

It is enough general to test equality on other values than records. We get a function equivalent to the program eq defined in section 2.3.1 by applying field_eq to the function id.

```
#let id_eq = field_eq id;;
id_eq : Pi (id : pre (α); χ) → Pi (id : pre (α); χ) → bool

#id_eq car truck;;
Typechecking error: collision between pre (string) and abs
```

The last example fails. This is not surprising since field is bound by a lambda in field_eq, and therefore its two instances have the same type, and so have both arguments x and y. In eq, the arguments x and y are independent since they are two instances of id. This is nothing else but ML generic polymorphism restriction. We emphasize that, as record polymorphism is entirely based on generic polymorphism, the restriction applies drastically to records.

2.3.3 Flexibility and Improvements

The method for typechecking records is very flexible: the operations on records have not been fixed at the beginning, but at the very end. They are parameters that can vary in many ways.

The easiest modification is changing the types of basic constants. For instance, asserting that $\{\_ \ with \ a = \_\}$ comes with type scheme:

$$\{\_ \ with \ a = \_\} : \Pi\,(a : abs \ ; \ \chi) \ \rightarrow \alpha \rightarrow \Pi\,(a : pre(\alpha) \ ; \ \chi)$$

makes the extension of a record with a new field possible only if the field was previously undefined. This slight change gives exactly the strict version that appears in both attempts to solve Wand's system [56, 86]. Weakening the type of this primitive may be interesting in some cases, because the strict construction may be easier to implement and more efficient.

We can freely change the types of primitives, provided we know how to implement them correctly. More generally, we can change the operations on records themselves. Since a defined field may not be dropped implicitly, it would be convenient to add a primitive removing explicitly a field from a record

$$\_ \setminus a : \Pi\,(a : \theta \ ; \ \chi) \ \rightarrow \Pi\,(a : abs \ ; \ \chi)\,,$$

In fact, the constant $\{\_ \ with \ a = \_\}$ is not primitive. It should be replaced by the strict version:

$$\{\_ \ with \ !a = \_\} : \Pi\,(a : abs \ ; \ \chi) \ \rightarrow \alpha \rightarrow \Pi\,(a : pre(\alpha) \ ; \ \chi)\,,$$

and the $\_ \setminus a$ constant, since the original version is the composition $\{\_ \setminus a \ with \ !a = \_\}$. Our encoding also allows typing a function that renames fields

$$rename^{a \leftarrow b} : \Pi\,(a : \theta \ ; \ b : \varepsilon \ ; \ \chi) \ \rightarrow \Pi\,(a : abs \ ; \ b : \theta \ ; \ \chi)$$

The renamed field may be undefined. In the result, it is no longer accessible. A more primitive function would just exchanges two fields

$$exchange^{a \leftrightarrow b} : \Pi\,(a : \theta \ ; \ b : \varepsilon \ ; \ \chi) \ \rightarrow \Pi\,(a : \varepsilon \ ; \ b : \theta \ ; \ \chi)$$

whether they are defined or not. Then the *rename* constant is simply the composition:

$$(\_ \setminus a) \circ exchange^{a \leftrightarrow b}$$

More generally, the decidability of type inference does not depend on the specific signature of the *pre* and *abs* type symbols. The encoding of records can be revised. We are going to illustrate this by presenting another variant for type-checking records.

We suggested that a good type system should allow some polymorphism on records values themselves. We recall the example that failed to type

```
#choice car truck;;
Typechecking error: collision between pre (string) and abs
```

because the age field was defined in car but undefined in truck. We would like the result to have a type with *abs* on this field to guarantee that it will not be accessed, but common, compatible fields should remain accessible. The idea is that a defined field should be seen as undefined whenever needed. From the point of view of types, this would require that a defined field with a value of type τ should be typed with both $pre(\tau)$ and abs.

Conjunctive types [32] could possibly solve this problem, but they are undecidable in general. Another attempt is to make abs of arity 1 by replacing each use of abs by $abs(\alpha)$ where α is a generic variable. However, it is not possible to write $\forall\theta\cdot\theta(\tau)$ where θ ranges over abs and pre. The only possible solution is to make abs and pre constant symbols by introducing an infix field symbol "." and write $abs.\alpha$ and $pre.\alpha$ instead of $abs(\alpha)$ and $pre(\alpha)$. It is now possible to write $\forall\varepsilon\cdot(\varepsilon.\tau)$. Formally, the signature $\mathcal{S}'$ is replaced by the signature $\mathcal{S}''$ given below, with a new sort *flag*:

$$\begin{array}{ll}
\mathcal{S}'' \vdash \Pi :: \mathit{field} \Rightarrow \mathit{type} & \\
\mathcal{S}'' \vdash \mathit{abs}^{\iota} :: \mathit{flag} & \iota \in \mathcal{K} \\
\mathcal{S}'' \vdash \mathit{pre}^{\iota} :: \mathit{flag} & \iota \in \mathcal{K} \\
\mathcal{S}'' \vdash .^{\iota} :: \mathit{flag} \otimes \mathit{type} \Rightarrow \mathit{field} & \iota \in \mathcal{K} \\
\mathcal{S}'' \vdash f^{Type} :: \mathit{type}^{\varrho(f)} \Rightarrow \mathit{type} & f \in \mathcal{C} \setminus \{\mathit{abs}, \mathit{pre}, .\} \\
\mathcal{S}'' \vdash (\ell^{L} : _\ ;\ _) :: \mathit{field} \otimes \mathit{field} \Rightarrow \mathit{field} & \ell \in \mathcal{L}, L \in \mathcal{P}_{fin}(\mathcal{L} \setminus \{\ell\})
\end{array}$$

Record constants now come with the following type schemes:

$$\begin{array}{c}
\{\,\} : \Pi\,(\mathit{abs}.\alpha) \\
_.a : \Pi\,(a : \mathit{pre}.\alpha\ ;\ \chi) \rightarrow \alpha \\
\{_\ \mathit{with}\ a = _\} : \Pi\,(a : \theta\ ;\ \chi) \rightarrow \alpha \rightarrow \Pi\,(a : \varepsilon.\alpha\ ;\ \chi)
\end{array}$$

Basic constants for ΠML′

It is easy to see that system ΠML′ is more general than system ΠML; any expression typeable in the system ΠML is also typeable in the system ΠML′: replacing in a proof all occurrences of abs by $abs.\alpha$ and all occurrence of $pre(\tau)$ by $pre.\tau$ (where α does not appear in the proof), we obtain a correct proof in ΠML′.

We show the types in the system ΠML′ of some of previous examples. Flag variables are written φ, θ and ψ. Building a record creates a polymorphic object, since all fields have a distinct *flag* variable:

```
#let car = {name = "Toyota"; age = "old"; id = 7866};;
car : Pi (name : φ.string; id : θ.num; age : ψ.string; abs.α)

#let truck = {name = "Blazer"; id = 6587867567};;
truck : Pi (name : φ.string; id : θ.num; abs.α)
```

Now these two records can be merged,

```
#choice car truck;;
it : Pi (name : φ.string; id : θ.num; age : abs.string; abs.α)
```

forgetting the age field in car. Note that if the presence of field age has been forgotten, its type has not: we always remember the types of values that have stayed in fields. Thus, the type system ΠML′ rejects the program:

```
#let person = {name = "Tim"; age = 31; id = 5656787};;
person : Pi (name : φ.string; id : θ.num; age : ψ.num; abs.α)

#choice person car;;
Typechecking error: collision between num and string
```

This is really a weakness of our system, since both records have common fields name and id, which might be tested on later. This example would be correct in the explicitly typed language QUEST [22]. If we add a new collection of primitives

$$\_ \setminus a : \Pi\,(a : \theta\ ;\ \chi) \rightarrow \Pi\,(a : \mathit{abs}.\alpha\ ;\ \chi)\,,$$

then we can turn around the failure above by explicitly forgetting label age in at least one record

```
#choice (car \ age) person;;
it : Pi (age : abs.num; name : φ.string; id : θ.num; abs.α)

#choice car (person \ age);;
it : Pi (age : abs.string; name : φ.string; id : θ.num; abs.α)

#choice (car \ age) (person \ age);;
it : Pi (age : abs.α; name : φ.string; id : θ.num; abs.β)
```

A more realistic example illustrates the ability to add annotations on data structures and type the presence of these annotations. The example is run into the system ΠML′, where we assume given an infix addition + typed with num → num → num.

```
#type tree (φ) = Leaf of num
#                | Node of {left: pre.tree (φ); right: pre.tree (φ);
#                           annot: φ.num; abs.unit}
#;;
New constructors declared:
Node : Pi (left : pre.tree (φ); right : pre.tree (φ); annot : φ.num; abs.unit) → tree (φ)
Leaf : num → tree (φ)
```

The variable φ indicates the presence of the annotation annot. For instance this annotation is absent in the structure

```
#let winter = 'Node {left = 'Leaf 1; right = 'Leaf 2 };;
winter : tree (abs)
```

The following function annotates a structure.

```
#let rec annotation =
# function
#    Leaf n → 'Leaf n, n
#  | Node {left = r; right = s} →
#     let (r,p) = annotation r in
#     let (s,q) = annotation s in
#       'Node {left = r; right = s; annot = p+q}, p+q;;
annotation : tree (φ) → tree (θ) * num

#let annotate x = match annotation x with y,_ → y;;
annotate : tree (φ) → tree (θ)
```

We use it to annotate the structure winter.

```
#let spring = annotate winter;;
spring : tree (φ)
```

We will read a structure with the following function.

```
#let read = function 'Leaf n → n | 'Node r → r.annot;;
read : tree (pre) → num
```

It can be applied to the value spring, but not to the empty structure winter.

```
#read winter;;
Typechecking error: collision between pre and abs
```

```
#read spring;;
it : num
```

But the following function may be applied to both winter and spring:

```
#let rec left =
# function
#    'Leaf n → n
# | 'Node r → left (r.left);;
left : tree (φ) → num
```

```
#left winter;;
it : num
```

```
#left spring;;
it : num
```

2.3.4 Extensions

In this section we describe two possible extensions. The two of them have been implemented in a prototype, but not completely formalized yet.

One important motivation for having records was the encoding of some object oriented features into them. But the usual encoding uses recursive types [20, 121]. An extension of ML with variant types is easy once we have record types, following the idea of [99], but the extension is interesting essentially if recursive types are allowed.

Thus it would be necessary to extend the results presented here with recursive types. Unification on rational trees without equations is well understood [54, 77]. In the case of a finite set of labels, the extension of theorem 5 to rational trees is easy. The infinite case uses an equational theory, and unification in the extension of first order equational theory to rational trees has no decidable and unitary algorithm in general, even when the original theory has one. But the simplicity of the record theory lets us conjecture that it can be extended with regular trees.

Another extension, which was sketched in [99], partially solves the restrictions due to ML polymorphism. Because subtyping polymorphism goes through lambda abstractions, it could be used to type some of the examples that were wrongly rejected. ML type inference with subtyping polymorphism has been first studied by Mitchell in [79] and later by Mishra and Fuh [44, 45]. The LET-case has only been treated in [55]. But as for recursive types, subtyping has never been studied in the presence of an equational theory. Although the general case of merging subtyping with an equational theory is certainly difficult, we believe that subtyping is compatible with the axioms of the algebra of record types. We discuss below the extension with subtyping in the finite case only. The extension in the infinite case would be similar, but it would rely on the previous conjecture.

It is straightforward to extend the results of [45] to deal with sorted types. It is thus possible to embed the language $\Pi\mathrm{ML}_{fin}$ into a language with subtypes $\Pi\mathrm{ML}_{\subset}$. In fact, we use the language $\Pi\mathrm{ML}'_{\subset}$ that has the signature of the language $\Pi\mathrm{ML}'$ for a technical reason that will appear later. The subtype relation we need is closed structural subtyping. Closed[2] structural subtyping is defined relatively to a set of atomic coercions as the smallest E-reflexive (i.e. that contains $=_E$) and

[2] In [45], the structural subtyping is *open*. With open structural subtyping only some of the atomic coercions are known, but there are potentially many others that can be used (opened) during typechecking of later phrases of the program. Closed subtyping is usually easier than open subtyping.

transitive relation $\subset$ that contains the atomic coercions and that satisfies the following rules [45]:

$$\frac{\sigma \subset \tau \qquad \tau' \subset \sigma'}{\tau \to \tau' \subset \sigma \to \sigma'}$$

$$\frac{\tau_1 \subset \sigma_1, \ldots \tau_p \subset \sigma_p}{f(\tau_1, \ldots \tau_p) \subset f(\sigma_1, \ldots \sigma_p)} \qquad f \in \mathcal{C} \setminus \{\to\}$$

In $\Pi\mathrm{ML}'_{\subset}$, we consider the unique atomic coercion $pre \subset abs$. It says that if a field is defined, it can also be view as undefined. We assign the following types to constants:

$$\begin{gathered}\{\} : \Pi\,(abs.\alpha_1, \ldots abs.\alpha_l) \\ \_.a : \Pi\,(\theta_1 \ldots,\ pre.\alpha \ldots \theta_l) \to \alpha \\ \{\_\ with\ a = \_\} : \Pi\,(\theta_1, \ldots \theta_l) \to \alpha \to \Pi\,(\theta_1 \ldots,\ pre.\alpha,\ \ldots \theta_l)\end{gathered}$$

Basic constants for $\Pi\mathrm{ML}'_{\subset}$

If the types look the same as without subtyping, they are taken modulo subtyping, and are thus more polymorphic. In this system, the program

```
let id_eq = field_eq id;;
```

is typed with:

$$\mathsf{id\_eq} : \{\mathsf{id} : \mathsf{pre}.\alpha;\ \chi\} \to \{\mathsf{id} : \mathsf{pre}.\alpha;\ \chi\} \to \mathsf{bool}$$

This allows the application modulo subtyping id_eq car truck. The field age is implicitly forgotten in truck by the inclusion rules. However, we still fail with the example choice person car. The presence of fields can be forgotten, yet their types cannot, and there is a mismatch between num and string in the old field of both arguments. A solution to this failure is to use the signature $\mathcal{S}'$ instead of $\mathcal{S}''$. However the inclusion relation now contains the assertion $pre(\alpha) \subset abs$ which is not atomic. Such coercions do not define a structural subtyping relation. Type inference with non structural inclusion has not been studied successfully yet and it is surely difficult (the difficulty is emphasized in [99]). The type of primitives for records would be the same as in the system $\Pi\mathrm{ML}_{fin}$, but modulo the non-structural subtyping relation.

Conclusion

We have described a simple, flexible and efficient solution for extending ML with operations on records allowing some sort of inheritance. The solution uses an extension of ML with a sorted equational theory over types. An immediate improvement is to allow recursive types needed in many applications of records.

The main limitation of our solution is ML polymorphism. In many cases, the problem can be solved by inserting retyping functions. We also propose structural subtyping as a more systematic solution. But it is not clear yet whether we would want such an extension, for it might not be worth the extra cost in type inference.

Acknowledgments

I am grateful for interesting discussions with Peter Buneman, Val Breazu-Tannen and Carl Gunter, and particularly thankful to Xavier Leroy and Benjamin Pierce whose comments on the presentation of this article were very helpful.

$$\text{If } \alpha \in \mathcal{V}(\tau) \wedge \tau \in e \setminus \mathcal{V}, \qquad \frac{U \wedge (\alpha \mapsto \sigma)(e)}{U \wedge \exists\, \alpha \cdot\ (e \wedge \alpha = \sigma)} \qquad \text{(Generalize)}$$

$$\frac{U \wedge a : \tau\ ;\ \tau' = abs = e}{U \wedge \bigwedge \begin{cases} abs = e \\ \tau = abs \\ \tau' = abs \end{cases}} \qquad \frac{U \wedge a : \alpha\ ;\ \alpha' = b : \beta\ ;\ \beta' = e}{U \wedge \exists\, \gamma \cdot \bigwedge \begin{cases} b : \beta\ ;\ \beta' = e \\ \alpha' = b : \beta\ ;\ \gamma \\ \beta' = a : \alpha\ ;\ \gamma \end{cases}} \qquad \text{(Mutate)}$$

$$\frac{U \wedge f(\tau_1, \ldots \tau_p) = f(\alpha_1, \ldots \alpha_p) = e}{U \wedge \bigwedge \begin{cases} f(\alpha_1, \ldots \alpha_p) = e \\ \tau_i = \alpha_i, \qquad i \in [1, p] \end{cases}} \qquad \text{(Decompose)}$$

$$\frac{U \wedge \alpha = e \wedge \alpha = e'}{U \wedge \alpha = e = e'} \qquad \text{(Fuse)}$$

Figure 2.1: Rewriting rules for record-type unification

Appendix

2.4 Unification on record types

The algorithm is an adaptation of the one given in [104], which we recommend for a more thorough presentation. It is described by transformations on unificands that keep unchanged the set of solutions. Multi-equations are multi-sets of terms, written $\tau_1 = \ldots \tau_p$, and unificands are systems of multi-equations, that is, multi-sets of multi-equations, with existential quantifiers. Systems of multi-equations are written U. The union of systems of multi-equations (as multi-sets) is written $U \wedge U'$ and $\exists\, \alpha \cdot U$ is the existential quantification of α in U. Indeed, $\exists$ acts as a binder and systems of multi-equations are taken modulo α-conversion, permutation of consecutive binders, and $\exists\, \alpha \cdot U$ is assumed equal to U whenever α is not free in U. We also consider both unificands $U \wedge \exists\, \alpha \cdot U'$ and $\exists\, \alpha \cdot U \wedge U'$ equal whenever α is not in U. Any unificand can be written $\exists\, W \cdot U$ where W is a set of variables, and U does not contain any existantial.

The algorithm reduces a unificand into a solved unificand in three steps, or fails. The first step is described by rewriting rules of figure 2.1. Rewriting always terminates. A unificand that cannot be transformed anymore is said completely decomposed if no multi-equation has more than one non-variable term, and the algorithm pursues with the occur check while instantiating the equations by partial solutions as described below, otherwise the unificand is not solvable and the algorithm fails.

We say that a multi-equation e' is inner a multi-equation e if there is at least a variable term of e' that appears in a non-variable term of e, and we write $e' < e$. We also write $U' \not< U$ for

$$\forall e' \in U',\ \forall e \in U,\ e' \not< e$$

The system U is independent if $U \not< U$.

The second step applies the rule

$$\text{If } e \wedge U \not\leq e, \qquad \frac{e \wedge U \rightsquigarrow}{e \wedge \hat{e}(U)} \qquad \text{(Replace)}$$

until all possible candidates e have fired the rule once, where $\hat{e}$ is the trivial solution of e that sends all variable terms to the non-variable term if it exists, or to any (but fixed) variable term otherwise. If the resulting system U is *independent* (i.e. $U \not\leq U$), then the algorithm pursues as described below; otherwise it fails and U is not solvable.

Last step eliminates useless existential quantifiers and singleton multi-equations by repeated application of the rules:

$$\text{If } \alpha \notin e \wedge U, \qquad \frac{\exists\, \alpha \cdot (\alpha = e \wedge U) \rightsquigarrow}{e \wedge U} \qquad \frac{\{\tau\} \wedge U \rightsquigarrow}{U} \qquad \text{(Garbage)}$$

This always succeeds, with a system $\exists\, W \cdot U$ that is still independent. A principal solution of the system is $\hat{U}$, that is, the composition, in any order, of the trivial solutions of its multi-equations. It is defined up to a renaming of variables in W. The soundness and correctness of this algorithm is described in [104].

The Replace step is actually not necessary, and a principal solution can be directly read from a completely decomposed form provided the transitive closure of the inner relation on the system is acyclic (see [104] for details).

With the signature $\mathcal{S}''$ the only change to the algorithm is the addition of the mutation rules:

$$\frac{a : \tau \;;\; \tau' = pre = e \rightsquigarrow}{\bigwedge \begin{cases} pre = e \\ \tau = pre \\ \tau' = pre \end{cases}} \qquad \frac{a : \alpha \;;\; \beta = \gamma_1.\gamma_2 = e \rightsquigarrow}{\exists\, \alpha_1\alpha_2\beta_1\beta_2 \cdot \bigwedge \begin{cases} \gamma_1.\gamma_2 = e \\ \alpha = \alpha_1.\alpha_2 \\ \beta = \beta_1.\beta_2 \\ \gamma_1 = a : \alpha_1.\beta_1 \\ \gamma_2 = a : \alpha_2.\beta_2 \end{cases}}$$

Note that in the first mutation rule, all occurrences of *pre* in the conclusion (the right hand side) of the rewriting rule have different sorts and the three equations could not be merged into a multi-equation. They surely will not be merged later since a common constant cannot fire fusion of two equations (only a variable can). As all rules are well sorted, rewriting keeps unificands well sorted.

Chapter 3

Projective ML

Ce chapitre a été publié dans [103].

Projective ML

Nous proposons un lambda-calcul projectif comme base pour exprimer les opérations sur les enregistrements. Les projections sont des enregistrements avec une valeur par défaut qui est comme projetée à l'infini. Le calcul étend le λ-calcul tout en conservant ses propriétés essentielles. Nous construisons le langage projective ML au dessus de ce calcul en ajoutant des liaisons polymorphes au λ-calcul projectif simplement typé. Nous montrons que projective ML possède la propriété d'autoréduction à la base de la sûreté de l'évaluation. Les projections sont une structure de donnée utile qui peut être compilée efficacement. De plus, les opérations habituelles sur les enregistrements peuvent être définies en termes de projections.

Projective ML

We propose a projective lambda calculus as the basis for operations on records. Projections operate on elevations, that is, records with defaults. This calculus extends lambda calculus while keeping its essential properties. We build projective ML from this calculus by adding the ML Let typing rule to the simply typed projective calculus. We show that projective ML possesses the subject reduction property, which means that well-typed programs can be reduced safely. Elevations are practical data structures that can be compiled efficiently. Moreover, standard records are definable in terms of projections.

Introduction

The importance of records in programming languages is commonly accepted. There have been many proposals for adding records in strongly typed functional languages [20, 119, 56, 86, 84, 99, 106, 27, 25, 49, 50]. However the topic is still active and there is not yet a best solution. Even for the most popular of them, ML, each implementation extends the core language with records of a very different kind.

For experts of record calculi, the multitude of works converges continuously towards a better comprehension of records, but it appears as a jungle of proposals for the novice that can hardly understand their very insidious differences. There is a lack of a simple formalism in which evaluation of row expressions could be described concisely and precisely. Furthermore, in a typed language, the typing rules often add technical restrictions that increase the confusion. This work started as a modest attempt to find a simple untyped record calculus in which most classical operations of records could be described. It ended in yet another proposal, but one that subsumes some others.

In the simplest view of records, there are only two operations. A record is a finite collection of objects, each component being addressed by name. The creation of a record takes as many name-object pairs as there are components and creates the corresponding record. The names used to address components are called labels; a label together with its component is a field. Reading information from a record takes a label that defines a field in the record and returns the component of that field. Thus the access of a component in a record should only require that the label does define a field in the record. Some type systems are more drastic, and require that the labels of all other fields of the records be also given at access time. This makes it impossible to use the same function to access the same field in two records having that field in common, but differing by other fields — a feature that is highly desirable.

The most popular extension of simple records is the creation of a record from another one by adding one field. This operation is called *record extension*. If the component may already be defined in the argument the extension is *free*, otherwise it is *strict*. Conversely, *record restriction* creates a record from another one by removing one of its field. As for extension, restriction can be free or strict.

The most difficult operation to type is still the *concatenation of records* that creates a record by combining the fields of two others [121, 50]. Again, record concatenation can be free or strict. There is also recursive concatenation that recursively merges the components of common fields, provided they are records themselves [86]. Record concatenation can be encoded with record extension, which gives one way of typechecking record concatenation [107]. However, none of the proposal for typing record concatenation is fully satisfactory.

Between extension and concatenation, there exists an intermediate operation that takes two records and a label and builds a record by copying all the fields of the first record except for the given label whose field is taken from the second one, whether it is defined or not. That is, either the label is undefined in both the second argument and the result or it is defined with the same value in both records. This operation, called *modification*, is strictly more powerful than extension and restriction, but much easier to type than concatenation, since it involves only one field. Other constructions, such as the exchange or renaming of fields are less popular, though they easily typecheck in some systems.

We introduce a projective lambda calculus as the basis for designing functional languages with records. In the first section, we study the Projective Lambda Calculus, written $P\Lambda$, extends the lambda calculus while preserving the Church-Rosser property. There is a simple projective type system for this calculus, for which the subject reduction theorem holds. In the second section, we

extend the simple projective type system with the ML Let typing rules and add concrete data types to the language: this defines the language we call projective ML. In the last section we elaborate on the significance of Projective ML from three different standpoints.

By lack of space, most of the proofs have been omitted, other are roughly sketched. See [103] for a more thorough presentation.

3.1 The projective lambda calculus

In this section we introduce the untyped projective lambda calculus. Then, we propose a simple type system for this calculus, we prove the subject reduction property and show that there are principal typings.

3.1.1 The calculus $P\Lambda$

The projective lambda calculus $P\Lambda$ is the lambda calculus extended with three constructions, namely the elevation, the modification and the projection. It is defined relatively to a denumerable collection of labels, written with letters a and b.

$$\begin{array}{lll} M ::= x & & \text{Variable} \\ \quad | \; \lambda\, x.\; M & & \text{Abstraction} \\ \quad | \; M\ M & & \text{Application} \\ \quad | \; [M] & & \text{Elevation} \\ \quad | \; M[a = M] & & \text{Modification} \\ \quad | \; M/a & & \text{Projection} \end{array}$$

The intended meaning of these constructions is given by the reduction rules of the projective lambda calculus. Namely, the rules are the classical β rule:

$$(\lambda\, x.\; M)\ N \longrightarrow M[x := N] \qquad (\beta)$$

plus the following projective rules (P):

$$[M]/a \longrightarrow M \qquad (\textsc{Default})$$
$$M[a = N]/a \longrightarrow N \qquad (\textsc{Access})$$
$$M[b = N]/a \longrightarrow M/a \qquad (\textsc{Skip})$$

As opposed to records, elevations can be projected on all labels.

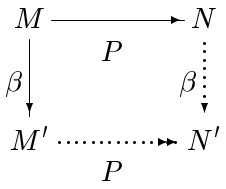
The compatible closure of $\longrightarrow$ is written $\longrightarrow\!\!\!\!\!\rightarrow$. The transitive closure of $\longrightarrow\!\!\!\!\!\rightarrow$ is written $\longrightarrow\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow$ and call βP-reduction.

Theorem 6 (Church-Rosser) *The calculus βP is Church-Rosser.*

This means that if M βP-reduces to N and N', then there exists a term M' such that both N and N' βP-reduces to M'.

Proof: The reductions β and P are Church-Rosser. The reduction P is a rewriting system that has no critical pair and is nœtherien, thus it is Church-Rosser. The reductions β and P commute,

since the diagram



commutes (this is checked by considering the relative positions of β- and P-redexes). ■

3.1.2 Projective types

Projective types extend the record types that have been introduced in [100, 104] in order to get a type system for the record extension of ML presented in [100, 106].

Record types are based on the idea that types of records should carry information on all fields saying for every label either the field is present or absent [99]. The way to deal with an infinite collection of labels is to give explicit information for a finite number of fields and gather all information about other fields in a template, called a row. Record types allow sharing between the same fields of two rows, but do not allow sharing between all fields of the same row (except for ground rows). When a type is coerced to a row, all projections must be shared for the same reason that lambda bound variables in ML cannot have polymorphic types.

$\tau ::=$		type	τ and σ
	α	type variable	α and β
	$\mid \tau \to \tau$	arrow type	
	$\mid \langle \rho \rangle$	projection type	
$\rho ::=$		row type	ρ and θ
	φ	row variable	φ and ψ
	$\mid \rho \Rightarrow \rho$	arrow row	
	$\mid a : \tau \ ; \ \rho$	defined row	
	$\mid \partial \, \tau$	shared row	

In fact, rows are sorted according to the set of labels that they cannot define. We omit this distinction here. The reader is referred to [104] for a more thorough presentation.

The equality on types is defined by the following axioms. *Left commutativity*:

$$a : \alpha \ ; \ (b : \beta \ ; \ \varphi) = b : \beta \ ; \ (a : \alpha \ ; \ \varphi)$$

simply means that the order of definition of rows does not matter. *Replication*:

$$\partial \, \alpha = a : \alpha \ ; \ \partial \, \alpha$$

means that shared rows are the same as rows defining the same type on all labels. Distributivity of arrows:

$$\partial \, \alpha \Rightarrow \partial \, \beta = \partial \, (\alpha \to \beta)$$

and

$$(a : \alpha \ ; \ \varphi) \Rightarrow (a : \beta \ ; \ \psi) = a : (\alpha \to \beta) \ ; \ (\varphi \Rightarrow \psi)$$

means that arrow rows are truly rows of arrows.

Lemma 2 *The theory of projective types is regular, unitary unifying and has a decidable unification algorithm.*

Hint: The regularity directly follows from the shape of the axioms. The theory of projective types is shown syntactic by extending the method developed in [104] for simple record terms. This is the difficult part of the proof. It is a consequence that the rewrite rules given in the appendix 3.4 are sound and complete. The termination of the algorithm is quite standard. Then, since the rewrite rules never introduce any disjunction, the theory is unitary. ■

The unification algorithm is described in the appendix 3.4.

3.1.3 A type system for $P\Lambda$

There are two kinds of typing judgements. A type assertion is the binding of a variable x to a type, written $x :^T \tau$ and a row assertion is the binding of a variable x to a row ρ, written $x :^R \rho$. A context is a list of assertions with rightmost priority. Mixed contexts contain both type and row assertions. Row contexts only contain row assertions. Concatenation of contexts is written by juxtaposition.

The judgement $H \vdash^T M : \tau$ means that in the mixed context H, the program M has type τ. The judgement $H, K \vdash^R M : \rho$ means that in the mixed context H and the row context K, the program M has row ρ. The first set of typing rules are the ones of the simply-typed lambda calculus:

$$\frac{x :^T \tau \in H}{H \vdash^T x : \tau} \qquad \text{T-Var}$$

$$\frac{H[x :^T \tau] \vdash^T M : \sigma}{H \vdash^T \lambda x.\, M : \tau \to \sigma} \qquad \text{T-Fun}$$

$$\frac{H \vdash^T M : \sigma \to \tau \qquad H \vdash^T N : \sigma}{H \vdash^T M\ N : \tau} \qquad \text{T-App}$$

The next set of rules deals with the elevations:

$$\frac{H \vdash^T M : \langle a{:}\,\sigma\ ;\ \rho\rangle \qquad H \vdash^T N : \tau}{H \vdash^T M[a = N] : \langle a{:}\,\tau\ ;\ \rho\rangle} \qquad \text{Modify}$$

$$\frac{H \vdash^T M : \langle a{:}\,\tau\ ;\ \rho\rangle}{H \vdash^T M/a : \tau} \qquad \text{Project}$$

$$\frac{H, \emptyset \vdash^R M : \rho}{H \vdash^T [M] : \langle\rho\rangle} \qquad \text{Elevate}$$

The first two rules are quite standard with record calculi. The last one describes the typing of an elevation. The elevated expression must be assigned a row. The row context shall binds variables that will be introduced during the typing of the current elevation, while previously bound variables are in the mixed context H. All expressions can be elevated, thus we need to assign rows to

applications and abstractions as well:

$$\frac{x :^R \rho \in K}{H, K \vdash^R x : \rho} \quad \text{R-Var}$$

$$\frac{H, K[x :^R \rho] \vdash^R M : \theta}{H, K \vdash^R \lambda x.\ M : \rho \Rightarrow \theta} \quad \text{R-Fun}$$

$$\frac{H, K \vdash^R M : \theta \Rightarrow \rho \qquad H, K \vdash^R N : \theta}{H, K \vdash^R M\ N : \rho} \quad \text{R-App}$$

Sometimes, one might get a type when a row is required. For instance, when a type derivation of $\lambda x.\ [x]$, the variable x will be assigned a type τ, but a row will be expected when typing x in the elevation. The type τ can be lifted to a shared row.

$$\frac{HK \vdash^T M : \tau}{H, K \vdash^R M : \partial\,\tau} \quad \text{Lift}$$

Conversely, a variable bound to the row $\partial\,(\tau)$ can be used with type τ:

$$\frac{x :^R \partial\,\tau \in H}{H \vdash^T x : \tau} \quad \text{Drop-Var}$$

Finally, since types are taken modulo E-equality:

$$\frac{H \vdash^T M : \sigma \qquad \sigma =_E \tau}{H \vdash^T M : \tau} \quad \text{T-Equal}$$

$$\frac{H \vdash^R M : \theta \qquad \theta =_E \rho}{H \vdash^R M : \rho} \quad \text{R-Equal}$$

We presented the previous set of rules (RT) since there are simple and very intuitive. There is a smaller and more regular set (S), given in the appendix 3.5, that are equivalent to the rules (RT). The judgements of (S) are $H, K \vdash^S : \rho$ were both H and K are row contexts (where superscript ^{R} is omitted).

Lemma 3 *The judgement $H \vdash^T M : \tau$ is derivable if and only if the judgement $H, \emptyset \vdash M : \partial\,\tau$ is derivable where $x :^T \tau$ in T is translated as $x : \partial\,\tau$ in S.*

Hint: The proof is by successive transformations of (RT) into equivalent systems ending with (S). The first step converts every type assertion $x :^T \tau$ in contexts into row assertions $x :^R \partial\,(\tau)$, replacing in the derivations, every occurrence of the rule T-Var by a rule Drop-Var. Rule T-Var is removed. The converse of the Lift rule:

$$\frac{H, K \vdash^R M : \partial\,\tau}{HK \vdash^T M : \tau} \quad \text{Drop}$$

is derivable in (RT), by an easy induction on the size of the derivation of the premise and by cases on the last rule of the derivation. It is added to (RT).

Successively, rules Fun and App are removed, record rules of (S) are added, then those of (RT) can be removed, rule Var is added and rule Drop-Var is removed. Last, Drop and Lift are shown to be useful only at the end of a derivation. ■

Lemma 4 (Stability by substitution) *Typings are stable by substitution.*

This property is quite immediate in the case of the a simple calculus.

The type inference problem is: given a triple $H, K \triangleright M : \rho$, find all substitutions μ such that $\mu(H), \mu(K) \vdash M : \mu(\rho)$. The type system (S) has principal typings if the set of solutions of every type inference problem is either empty, or has a maximal element called a principal solution, and if, in addition, there exists an algorithm that takes a type inference problem as input and returns a principal solution or an indication of failure if no solution exists.

Theorem 7 (Principal typings) *The type system of $P\Lambda$ has principal typings.*

Hint: Type inference for $P\Lambda$ is in the general framework of extending the ML type system with an equational theory on types. The comma that splits the contexts into two parts is a detail, since the system (S) is still syntax directed. The principal type property for such a system holds in general whenever the axiomatic theory on types is regular, unitary unifying and as a decidable unification algorithm [100].

Type inference is based on the syntacticness of the theory of projective types and the unification algorithm that follows. It proceeds exactly as for the language with record extension presented in [106]. ■

The algorithm for type inference can be found in the Appendix 3.6 for the language PML presented in the next section.

3.1.4 Subject reduction

Subject reduction holds if reduction preserves typings: for any program M and N, if M has type τ in the context H, K and βP-reduces to N, then N has type τ in context H, K.

Theorem 8 (Subject reduction) *Subject reduction holds in $P\Lambda$.*

Hint: It is shown independently for all cases of reduction at the root, then it easily follows for deeper reductions. The difficult case is ELEVATE. It uses the lemma *if $HK, \emptyset \vdash M : (a : \tau; \theta)$ is derivable in S, then so is $HK, \emptyset \vdash M : \partial\tau$* which is proved with a little stronger hypothesis by induction on the length of the derivation of the premise and cases on the last rule that is not an equality rule. ■

3.2 The language PML

Since the simply typed projective lambda calculus behaves nicely, we extend it to a full language, PML, in two steps. We add the ML *Let* typing rule and then concrete data types. In each case we check that the principal type property and subject reduction still hold.

3.2.1 Let polymorphism

We extend the projective calculus with a *let* construction

$$M ::= \ldots \mid let\ x = M\ in\ N$$

The *let* is syntactic sugar for marked redexes

$$(\lambda\, x.\ N)^*\ \ N$$

Thus, there is no special reduction rule for *let* redexes but the (β) rule:

$$(\lambda\, x.\ M)^*\ \ N \longrightarrow (x \mapsto N)(M) \qquad (\beta)$$

Therefore the calculus remains Church-Rosser.

Types are extended with type schemes. Type schemes are pairs of a set of variables and a type or a row, written $\forall\, W \cdot \tau$ or $\forall\, W \cdot \rho$. Formally, variables should be annotated with their sorts, but the sorts can be recovered from the occurrences of variables in their scheme. We identify type schemes modulo α-conversion of bound variables, and elimination of quantification over variables that are not free.

Type assertions now bind variables to type schemes. The rules VAR are changed to:

$$\frac{x : \forall\, W \cdot \rho \in K \qquad dom\,(\mu) \subset W}{H, K \vdash x : \mu(\rho)}$$

$$\frac{x : \forall\, W \cdot \partial\, \tau \in H \setminus K \qquad dom\,(\mu) \subset W}{H, K \vdash x : \partial\, \mu(\rho)}$$

The LET rule is

$$\frac{H, K \vdash M : \rho \qquad H, K[x : \mathcal{V}(\rho) \setminus \mathcal{V}(HK)] \vdash N : \theta}{H, K \vdash \mathit{let}\ x = M\ \mathit{in}\ N : \theta} \qquad \textsc{Let}$$

where $\mathcal{V}(\rho)$ is the set of free variables in ρ and $\mathcal{V}$ is naturally extended to contexts.

The extension of $P\Lambda$ with *let* binding does not interfere with projections, and the substitution lemma, and the principal typing property and subject reduction theorems easily extend to PML.

3.2.2 Concrete data types

The language is now parameterized by a finite collection of concrete data types. For sake of simplicity, we consider a single two-constructor data type. We shall make other simplifying assumptions on types below, but it is possible to generalize to arbitrary data types.

The data type that we consider could be declared in ML as:

$$\text{type}\ bar\,(\rho) = A \mid B\ \text{of}\ \rho$$

The syntax is extended with:

$$\begin{aligned} M ::= {} & \ldots \\ & \mid A \mid B(M) \\ & \mid \mathit{match}\ M\ \ \mathit{with}\ A \Rightarrow M \mid B(y) \Rightarrow M \end{aligned}$$

The new reduction rules are:

$$\begin{aligned} &(\mathit{match}\ A\ \mathit{with}\ A \Rightarrow M \\ &\qquad\qquad \mid B(y) \Rightarrow N) \longrightarrow M \\ &(\mathit{match}\ B(L)\ \mathit{with}\ A \Rightarrow M \\ &\qquad\qquad \mid B(y) \Rightarrow N) \longrightarrow (\lambda\, y.\ N)\ L \end{aligned}$$

These δ-reductions are CR and commute with βP. Therefore the language PML with sums is still Church-Rosser.

Types are also extended with a symbol *bar* of arity one.

$$\begin{array}{lll} \tau ::= & \ldots & \text{Old type} \\ & bar\,(\tau) & \text{bar type} \\ \rho ::= & \ldots & \text{Old row} \\ & bar\,(\rho) & \text{bar row} \end{array}$$

We should have used two different symbols for bar types and bar rows, but the context will distinguish them. The symbol *bar* obeys the two distributivity axioms:

$$bar\,(a : \alpha; \varphi) = a : bar\,(\alpha); bar\,(\varphi)$$
$$\partial\,(bar\,(\alpha)) = bar\,(\partial\,\alpha)$$

We add the three typing rules:

$$\frac{}{H, K \vdash A : bar\,(\rho)} \qquad \frac{H, K \vdash M : \rho}{H, K \vdash B(M) : bar\,(\rho)}$$

$$\frac{\begin{array}{c} H, K \vdash L : bar\,(\theta) \\ H, K \vdash M : \rho \qquad H, K \vdash \lambda\, y.\ N : \theta \Rightarrow \rho \end{array}}{match\ \ L\ \ with\ \ A \Rightarrow M \mid B(y) \Rightarrow N : \rho}$$

Theorem 9 *The language PML with sums has principal typings.*

Theorem 10 *Subject reduction holds for PML with sums.*

3.3 The three views of PML

Projective ML is a practical language of records with default values. It is also a language in which all operations of classical records but concatenation are definable. Finally, computation inside elevations introduces a new kind of polymorphism.

3.3.1 Records with default values

To the author's knowledge, this feature has never been introduced in the literature before. Instead of starting with empty records that can be extended with new fields, projective ML initially creates records with the same default value on all fields. Then a finite number of fields can be modified. Thus, all fields are always defined and can be read.

The introductory examples below have been typechecked by a prototype typechecker written in Caml-Light [67]. The first examples are:

```
#type unit = Unit;;
#let r = [Unit];;
r : shared [unit]

#r/a;;
it : shared unit

#type bool = True | False;;
```

```
#let s = r [a = True];;
s : shared [a : bool; unit]

#s/a;;
it : shared bool
```

The a field of s cannot be removed, but it can be reset to its default value. Whenever the types of fields are known statically, but not their presence, the attendance can be dynamically checked:

```
#type field (φ) = Absent | Present of φ;;
#let r = [Abs] [a = Present (True)]
#                  [b = Present (Unit)];;
r : shared [a : field (bool); b : field (unit); field (φ)]

#let check x =
#  match x with Present y ⇒ y
#       | Absent ⇒ failwith "Absent field";;
check : field (φ) ⇒ φ

#let v = check (r/a);;
v : shared bool
```

If the presence of fields is stactically known, the two-constructor data type can be replaced by two one-constructor data types, leaving the typechecker check attendances.

```
#type absent = Absent;;
#type present (φ) = Present (φ);;
#let get x = match x with Present y ⇒ y;;
get : present (φ) ⇒ φ

[Absent][a = Present (true)][b = Present (unit)];;
it : shared [a : present (bool); b : present (unit); absent]

#let v = get (it/a);;
v : shared bool
```

Record with defaults are not just an untractable toy feature. They can be compiled very efficiently, as classical records [101].

3.3.2 Classical records

Continuing the example above, we show that classical records are definable in projective ML. Precisely, classical record operations are just syntactic sugar for:

$$\begin{aligned} \{\} &\equiv [Absent] \\ \{M \ with \ a = N\} &\equiv M[a = Present \ (N)] \\ (M.a) &\equiv get \ (M/a) \end{aligned}$$

Many other constructions are programmable as well, since projective ML allows the manipulation of fields whether they are present or absent.

$$\begin{aligned} M \setminus a &\equiv M[a = Absent] \\ \{M \ but \ a \ from \ N\} &\equiv M[a = N/a] \\ \{exchange \ a \ and \ b \ in \ M\} &\equiv let \ u = M/a \ in \\ &\quad\ let \ v = M/b \ in \\ &\quad\ M[a = v][b = u] \end{aligned}$$

Though efficiency is not our main goal here, it is important to emphasize that dealing explicitly with the presence of fields does not cost anything. Since both *abs* and *pre* data types have unique constructors, the constructors need not be represented explicitly. That is, the presence of fields can be statically computed by the typechecker. Even the default value *Absent* need not be represented, since it is the only value in its type. Thus the (very small) overhead for computing with elevations only costs when there are used.

Obviously, the projective implementation of standard records can be packed in an abstract data type or a module so that the two types *pre* and *abs* and their constructors are not visible outside, and the presence of fields cannot be manipulated by hand. But elevations and projections will remain visible, can be used whenever defaults values in records are desirable, or also to implement another variant of classical records.

3.3.3 Projection polymorphism

The last view of projective ML is quite unexpected. The elevations are assigned rows that are in fact "template" types. That is, they can be read on any component by taking a copy of the template; therefore the type of two projections will not be equal but isomorphic. For instance, with classical records as in [106] (or using the syntactic sugar of the previous section) the function that reads the a field of a record has type:

$$[a : pre\,\tau; \varphi] \to \tau$$

But this type can also be seen as[1]:

$$[a : pre\,\tau; b : \alpha; \psi] \to \tau$$

With classical records, this polymorphism allows the finite representation of a potentially infinite product of types, and nothing more. In projective ML, we can fill the elevations with any value and even compute inside. The identity function elevation $[\lambda\, x.\ x]$ has type $[\varphi \Rightarrow \varphi]$. Taking its projection on two arbitrary fields gives twice the same value but with two isomorphic types $\alpha \to \alpha$ and $\beta \to \beta$. The program,

$$(\lambda\, x.\ x\ \ x)\ \ (\lambda\, x.\ x) \tag{1}$$

cannot be written in ML without a LET. In projective ML one can write:

$$(\lambda\, x.\ x/a\ \ x/b)\ \ [\lambda\, x.\ x] \tag{2}$$

which has type $\alpha \to \alpha$. It can be argued that this is not exactly the same program, and that, if program transformations are allowed, then the following ML program also computes the same result.

$$(\lambda\, xy.\ x\ \ y)\ \ (\lambda\, x.\ x)\ \ (\lambda\, x.\ x) \tag{3}$$

This is certainly true, but the program (3) is much bigger than the program (1) and duplicates some of the code. The expression (2) is almost as small as the expression (1) and takes less time to typecheck (for bigger example of course, since all examples here are too small to allow any comparison). In (3), the body of $\lambda\, x.\ x$ is typed twice, but it is typed only once in (2) before the resulting type is duplicated by unification.

[1] In [104] we define canonical forms and show that both type have the same canonical form, though they are not equal (the latter is less general).

Moreover, if we consider a variant of PML without the possibility of modifying elevations,

$$M ::= x \mid \lambda\, x.\ M \mid M\ M \mid [M] \mid M/a$$

then projections always access the default value of elevations (since they could not be modified). Elevation and projection can both be implemented as empty code. They only modifies the types (they are called retyping functions), and helps the typechecker as if they were type annotations. The elevation indicates that an expression may be used later with different types, and thus should be typed with a row. The projection requires the use of a copy of the row template instead of the row itself. The copy is kept inside the row for constraint propagation.

Breaking the expression (2), the subexpression $(\lambda\, x.\ x/a\ \ x/b)$ has type:

$$[a : \tau \to \sigma; b : \tau] \to \sigma \tag{4}$$

There are obvious similarities with conjunctive types [32, 91]. This expression would have the conjunctive type

$$(\tau \to \sigma \wedge \tau) \to \sigma \tag{5}$$

Projective ML differs from conjunctive types by naming the conjunctions, but also in some deeper way. The projection, which correspond to the expansion in conjunctive types, is much more restrictive than the expansion. An interesting comparison would be with the decidable restriction of conjunctive types that has been recently proposed by Coppo and Denzianni [31].

There is an important limitation in the type system of projective ML: it is a two-level design. Elevations inside elevations get typed with shared rows and projective polymorphism is lost. A stratified version with types, rows, rows of rows, etc. composing an infinite row tower can be imagined. The author has actually worked on such a version but has not proved yet that it is correct.

Another form of this limitation of projective polymorphism is its failure to cross elevations. The best type for $\lambda\, x.\ [x]$ is $\partial\, \alpha \Rightarrow [\partial\, \alpha]$, while we would expect $\varphi \Rightarrow [\varphi]$. Variables in elevations that are bound outside of the current elevation in which they appear can only have shared rows.

Projective polymorphism combines nicely with generic polymorphism. The two concepts are orthogonal. Here is an example that combines both:

$$\begin{array}{l} \mathit{let}\ F = \lambda\, f.\ \lambda\, x, y.\ f/a\ \ x, f/b\ \ y\ \mathit{in} \\ F\ \ [I]\ \ (I, K), F\ \ [K]\ \ (I, K) \end{array}$$

where I and K are abbreviations for $\lambda\, x.\ x$ and $\lambda\, xy.\ x$. It is typeable in projective ML.

Conclusions

We have introduced Projective ML, and shown that it is a type-safe language. Projective ML exceeds ML on two opposite fields.

- Elevations, modifications and projections are extensible records with defaults. With only three operations that can be compiled very efficiently, they provide the ML language with enough power to define all variants of classical records.
- Projective ML brings in the type system a restricted form of conjunctive polymorphism.

The curiosity of Projective ML is that both features are almost independent but one still need the other. The most intriguing of the two is projective polymorphism, for which more investigation is still needed.

Appendix

3.4 Unification on projective types

We describe the unification algorithm by transformation rules on unificands (multi-sets of equations). The formalism is the one of [104] in general, improved with existential unificands [57]. A multi-equation is a multi-set of terms written $\tau_1 \doteq \dots \tau_n$. A solution of a multi-equation is a substitution that unifies all the terms of the multi-equation. A multi-set of multi-equations is noted $U_1 \wedge \dots U_p$. Its solutions are the substitutions that satisfy all the multi-equations. We also use existential unificands, written $\exists\alpha.U$, whose solutions are the restrictions of the solutions of U on variables distinct from α. Indeed, $\exists$ acts as a binder, and existential unificands are equal modulo α-conversion. Consecutive binders can be exchanged, and $\exists\alpha.U$ is equal to U whenever α is not free in U. We identify unificands modulo the previous equalities.

Two unificands U and U' are *equivalent*, and we write $U \equiv U'$ if they have the same set of solutions. The relation $\equiv$ is obviously an equivalence. It is also a congruence, that is, parts of unificands can be replaced by equivalent parts. We also write $\bot$ and $\top$ for unificands that are respectively equivalent to the empty set and the set of all substitutions.

The input of the unification algorithm is a multi-set of equations. The output will be failure or a most general solution of the input unificand. It proceeds in three steps. All of these steps are described by transformations of unificands that are equivalences.

Most of the transformations are valid for both types and rows. We write χ and ξ for terms and π for variables that can be of both kinds. The first step is the generalization:

$$\frac{e \doteq \chi[\xi]}{\exists\,\pi.\, e \doteq \chi[\pi] \wedge \pi \doteq \xi} \qquad \textsc{Generalize}$$

An iteration of this rule will transform any system into one that contains only small terms (terms of height at most one).

The second step is only defined on small unificands, and keeps them small. The mutation of unificands is one of the four following transformations (f is a symbol of arity p and I is the segment

of integers $[1, p]$):

$$\frac{a : \tau; \rho \doteq f\,(\theta_i)_I}{\exists\,(\alpha_i)_I\;(\varphi_i)_I\,.\;\bigwedge\left\{\begin{array}{l}\tau \doteq f\,(\alpha_i)_I\\ \rho \doteq f\,(\varphi_i)_I\\ \theta_i \doteq a : \alpha_i; \varphi_i \quad i \in I\end{array}\right.} \qquad \textsc{Mut}_{a \triangleright f}$$

$$\frac{a : \tau; \rho \doteq b : \sigma; \theta}{\exists\,\varphi.\;\bigwedge\left\{\begin{array}{l}\rho \doteq b : \sigma; \varphi\\ \theta \doteq a : \tau; \varphi\end{array}\right.} \qquad \textsc{Mut}_{a \triangleright b}$$

$$\frac{\partial\,(\tau) \doteq a : \sigma; \rho}{\exists\,\alpha.\;\bigwedge\left\{\begin{array}{l}\alpha \doteq \sigma \doteq \tau\\ \rho \doteq \partial\,(\alpha)\end{array}\right.} \qquad \textsc{Mut}_{\partial \triangleright b}$$

$$\frac{\partial\,(\tau) \doteq f\,(\rho_i)_I}{\exists\,(\alpha_i)_I\,.\;\bigwedge\left\{\begin{array}{l}\tau \doteq f\,(\alpha_i)_I\\ \rho_i \doteq \partial\,(\alpha_i) \quad i \in I\end{array}\right.} \qquad \textsc{Mut}_{f \triangleright \partial}$$

For all other pairs of terms (χ, ξ), if they have identical top symbols, they are decomposable, that is

$$\frac{\chi \doteq \xi}{\bigwedge_I (\chi_{/i} \doteq \xi_{/i})} \qquad \textsc{Decompose}$$

otherwise they produce a collision

$$\frac{\chi \doteq \xi}{\bigwedge_I (\chi_{/i} \doteq \xi_{/i})} \qquad \textsc{Collision}$$

All mutation, decomposition and collision rules can be generalized to rules where the premise is a multi-equation rather than an equation: for any mutation rule

$$\frac{\chi \doteq \xi}{Q}$$

we build the generalized mutation rule:

$$\frac{e \doteq \chi \doteq \xi}{e \doteq X \wedge Q}$$

The fusion of multi-equations is:

$$\frac{\pi \doteq e \wedge \pi \doteq e'}{\pi \doteq e \doteq e'} \qquad \textsc{Fuse}$$

Applying the generalized mutation and the fusion in any order always terminates on small unificands. Unificands that cannot be reduced are necessarily in *canonical* forms, that is, completely decomposed and fused.

The last step does the occur check on canonical unificands while instantiating the equations by partial solutions. On canonical unificands Q, we say that the multi-equation e' is directly inner the multi-equation e if there is at least a variable term of e' that appears in a non variable term of e. We note $\leqslant_Q$ its transitive closure. The occur check is the rule

$$\text{if } e \leqslant_Q e, \qquad \frac{Q}{\bot} \qquad \textsc{Occur}$$

Otherwise, we can apply the rule:

$$\text{if } e \not\leqslant Q, \qquad \frac{e \wedge Q}{e \wedge \hat{e}(Q)} \qquad \textsc{Replace}$$

where $\hat{e}$ is the trivial solution of e that sends all variable terms of e to the non variable term if it exists, or to any variable term otherwise. The Replace rule is completed by the elimination of useless existentials

$$\text{if } \pi \notin e \cap Q, \qquad \frac{\exists\, \pi.\, (\pi \doteq e \wedge Q)}{e \wedge Q} \qquad \textsc{Restrict}$$

The succession of the three steps either fails or ends with a system $\exists W.Q$ where all multi-equations are independent. A principal solution of the system is $\hat{Q}$, that is, the composition, in any order, of the trivial solutions of its multi-equations. It is defined up to a renaming of variables in W.

The last step may be reduced to the occur check, and the equations in the unificand need not be instantiated by rule Replace, since the canonical unificand itself is a good and compact representation of a principal unifier.

Although it is described in a more general framework, the algorithm is very close to the one of Martelli-Montanari for empty theories [77], some of the collisions have been replaced by mutations in a way that copies the axioms of the theory. This is a property of syntactic theories [61, 62]. Proving the correctness of the algorithm is reduced to proving the syntacticness of the theory and the termination of the second step. Proving the termination is standard, but proving that the theory is syntactic is the difficult part.

The second step may not be restricted to small terms. In this case the generalized mutation and decomposition rules need to include the minimum of generalization so that there is enough sharing to ensure the termination.

3.5 A simpler set of typing rules for the projective calculus

The judgements are of the form $H, K \vdash M : \rho$, where H and K are row assertions. The typing rules, called (S) are:

$$\frac{x : \partial\,\tau \in H \setminus K}{H, K \vdash x : \partial\,\tau} \qquad \frac{x : \rho \in K}{H, K \vdash x : \rho} \quad \textsc{Var}$$

$$\frac{H, K[x : \rho] \vdash M : \theta}{H, K \vdash \lambda\, x.\ M : \rho \Rightarrow \theta} \quad \textsc{Fun}$$

$$\frac{H, K \vdash M : \theta \Rightarrow \rho \qquad H, K \vdash N : \theta}{H, K \vdash M\ N : \rho} \quad \textsc{App}$$

$$\frac{HK, \emptyset \vdash M : \rho}{H, K \vdash [M] : \partial\,\langle \rho \rangle} \quad \textsc{Elevate}$$

$$\frac{\begin{array}{c} H, K \vdash N : \partial\,(\sigma) \\ H, K \vdash M : \partial\,\langle a{:}\,\tau\ ;\ \theta \rangle \end{array}}{H, K \vdash M[a = N] : \partial\,\langle a{:}\,\sigma\ ;\ \theta \rangle} \quad \textsc{Modify}$$

$$\frac{H, K \vdash [M] : \partial\,\langle a{:}\,\tau\ ;\ \theta \rangle}{H, K \vdash M/a : \partial\,\tau} \quad \textsc{Project}$$

$$\frac{H, K \vdash M : \theta \qquad \theta =_E \rho}{H, K \vdash M : \rho} \quad \textsc{Equal}$$

3.6 Type inference

The above set of rules is completed with:

$$\frac{\begin{array}{c} H, K \vdash M : \rho \\ H, K[x : \forall\ (\mathcal{V}(\rho) \setminus \mathcal{V}(HK)) \cdot \rho] \vdash N : \theta \end{array}}{H, K \vdash \mathit{let}\ x = M\ \mathit{in}\ N : \theta} \quad \textsc{Let}$$

The rules are not exactly those of ML. The two rules Modify and Project can be treated as application of constants. The rule equal, due to an extended type equality, does not add any difficulty, provided that the theory is regular and has a decidable and unitary unification algorithm [100]. The only difference with ML (extended with equations on types) is the mark in the context. However, the position of the mark is rigid, and the type inference algorithms of ML very easily extends to the system S. We describe the algorithm in terms of unificands. The substitution lemma (that extends to PML) allows to consider type inference problems as unificands, written $H, K \triangleright M : \rho$, whose solutions are the substitutions μ such that $\mu(H), \mu(K) \vdash M : \mu(\rho)$ is a valid judgement. We give below equivalence transformations of these unificands.

Case Var**:** If $x : \partial\,\tau$ is in $H \setminus K$, and μ is a renaming of variables of $\mathcal{V}(\tau)$ outside of σ, then

$$\frac{H, K \triangleright x : \sigma}{\exists\, \mathcal{V}(\mu(\tau)).\ \sigma = \mu(\tau)} \quad \text{T-Var}$$

If $x : \rho$ is in K, and is a renaming of variables of $\mathcal{V}(\rho)$ outside of θ, then

$$\frac{H, K \triangleright x : \theta}{\exists\, \mathcal{V}(\mu(\rho)).\, \theta = \mu(\rho)} \quad \text{R-VAR}$$

If x is not in HK, then $H, K \triangleright x : \alpha$ is not solvable.

Case APP**:**

$$\frac{H, K \triangleright MN : \rho}{\exists\, \psi.\, H, K \triangleright M : \psi \wedge H, K \triangleright N : \psi \Rightarrow \rho} \quad \text{APP}$$

Case FUN**:**

$$\frac{H, K \triangleright \lambda\, x.\ M : \rho}{\exists\, \varphi\psi.\, H, K[x : \varphi] \triangleright M : \psi \wedge \rho = \varphi \Rightarrow \psi} \quad \text{FUN}$$

Case LET**:** If β is outside of HK and $\exists\, W.\, Q$ is a solvable independent unificand equivalent to $H, K \triangleright M : \beta$, then

$$\frac{H, K \triangleright \mathit{let}\ x = M\ \mathit{in}\ N : \tau}{\exists\, W.\, H, K[x : \hat{Q}(\beta)] \triangleright N : \tau} \quad \text{LET}$$

If $H, K \triangleright M : \beta$ is not solvable, then neither is $H, K \triangleright \mathit{let}\ x = M\ \mathit{in}\ N : \alpha$.

Case ELEVATE**:**

$$\frac{H, K \triangleright [M] : \rho}{\exists\, \alpha.\, HK, \emptyset \triangleright M : \alpha \wedge \partial\, \alpha \doteq \rho} \quad \text{ELEVATE}$$

The above rules applied in any order either fail or reduce any type inference problem to a unification problem.

Chapter 4

Typage de la concaténation des enregistrements à l'œil

Ce chapitre a été publié dans [107].

Typage de la concaténation des enregistrements à l'oeil

Nous montrons que dans un langage fonctionnel avec des enregistrements extensibles la concaténation des enregistrements est gratuite. Nous donnons une traduction de la concaténation en utilisant une opération d'extension. Nous obtenons un système de type pour un langage avec la concaténation en composant la traduction avec le typage de l'extension des enregistrements. Nous appliquons cette méthode à une version de ML avec une opération d'extension. Nous obtenons une extension simple et flexible de ML avec une opération de concaténation symétrique ou asymétrique qui possède un algorithme de synthèse des types efficace en pratique. Pour obtenir dans le langage avec concaténation une opération d'effacement des champs, il faut ajouter une nouvelle opération aux enregistrements extensibles.

Les langages à objets bénéficient de ce codage puisqu'il montre que l'héritage multiple n'a pas en fait besoin de la concaténation des enregistrements mais seulement d'une opération d'extension.

Typing Record Concatenation for Free

We show that any functional language with record extension possesses record concatenation for free. We exhibit a translation from the latter into the former. We obtain a type system for a language with record concatenation by composing the translation with typechecking in a language with record extension. We apply this method to a version of ML with record extension and obtain an extension of ML with either asymmetric or symmetric concatenation. The latter extension is simple, flexible and has a very efficient type inference algorithm in practice. Concatenation together with removal of fields needs one more construct than extension of records. It can be added to the version of ML with record extension. However, many typed languages with records cannot type such a construct. The method still applies to them, producing type systems for record concatenation without removal of fields. Object systems also benefit from the encoding which shows that multiple inheritance does not actually require the concatenation of records but only their extension.

Introduction

Dictionaries are an important data abstraction in programming languages. They are basically partial functions from keys to values. A simple implementation of dictionaries is the *association list*, commonly called *A-list* . A-lists are lists of pairs, the first component being the key to access the value of the second component. The usual *cons* and *append* operations provide facilities for extending the domain of an A-list and merging two A-lists into one defined on the union of the domains of the input lists, respectively. Access to a given key may fail when the key is not in the domain of the A-list, which cannot be checked statically. Records are a highly restricted form of A-lists. Keys may no longer be any values, but belong to a distinguished set of atomic values, called labels. All fields of a record must be specified at creation time. These restrictions make it possible to perform static checks on accesses to record fields.

Then, an important goal in typechecking records, was to allow a record with many fields to be used instead of a records with fewer fields. This was first suggested by Cardelli in the language Amber [21] using inclusion on monomorphic types.

Later, Wand [119] used polymorphism instead of a specific inclusion relation on types. He also re-imported the *cons* operation of A-lists which became the extension of records with new fields. Originally, this construction was free (existing fields could be redefined), but strict versions (existing fields could not be redefined) have been proposed [86, 56] to avoid typechecking difficulties. Note that *cons* on A-lists naturally implements free extension.

Record extension quickly became popular, but many languages still only provide the strict version [56, 84, 49]. Finally Wand re-imported the *append* of A-lists, calling it record concatenation. An important motivation for this is the encoding of multiple inheritance [121] in object oriented languages.

Record concatenation is still considered a challenge, since it is either very restricted [49] or leads to combinatorial explosion of typechecking [120]. We propose a general approach to concatenation. In fact we claim that concatenation comes for free once record extension is provided. We justify this assertion by presenting an encoding of the latter into the former. The interest of the encoding is to provide a type system for record concatenation by composing the coding with a type system for record extension.

We introduce the translation in an untyped framework in section 4.1. In section 4.2, we apply it to an extension of ML for record extension. In the last section we briefly illustrate the encoding on a few other languages.

4.1 Encoding of concatenation

In this section we describe how concatenation can be encoded with extension. The language with record extension, L, is an extension of the untyped λ calculus plus distinguished constructs for record expressions:

$M ::= x$	variable
$\mid \lambda x.\, M$	abstraction
$\mid M\ M$	application
$\mid \{\}$	empty record
$\mid \{M \text{ with } a = M\}$	record extension
$\mid M.a$	record access

The semantics of records is the usual one. Informally, they are partial functions from labels to values. The empty record is defined nowhere. Accessing a field of a record is applying the record to that field. It produces an error if the accessed field is not defined. The *free* extension of a record with a new field defines or redefines that field with the new value. The *strict* extension does the same if the field was undefined, but produces an error otherwise. In an untyped language the free extension is preferred since the more well typed programs, the better.

The *concatenation* (or *merge*) operator $\|$ takes two records and returns a new record composed of all fields defined in any of its arguments. There are different semantics given to the merge, when both records define the same field: *symmetric* concatenation rejects this case [50] while *asymmetric* concatenation takes the value from the last record [121]. We will not consider *recursive* concatenation that would compute the concatenation of common fields by recursively concatenating their values.

The language with record concatenation, $L^{\|}$ is

$$M ::= x \mid \lambda\, x.\ M \mid M\ M \mid \{\} \mid \{a = M\} \mid M \parallel M \mid M.a$$

The language is an extension of L with a construct for concatenation, but record extension has been replaced by one-field records that are more primitive in the presence of concatenation, since[1]:

$$\{M \text{ with } a = N\} \equiv M \parallel \{a = N\}$$

Reading this equality from right to left is also interesting: it means that one-field concatenation can be written with record extension only. It gives the expected semantics of asymmetric concatenation when the extension is free and the semantics of symmetric concatenation when the extension is strict. We are going to generalize this to a translation from the language $L^{\|}$ to the language L.

4.1.1 The untyped translation

The following translation works for both asymmetric and symmetric concatenation. We arbitrarily choose asymmetric concatenation.

The extension of fields provides the one-field concatenation operation:

$$\lambda\, r.\ (r \parallel \{a = M\}) = \lambda\, r.\ \{r \text{ with } a = M\},$$

which we write $\{a = M\}^{\dagger}$. In fact, we can compute $r \parallel s$ whenever we know exactly the fields of s, since

$$r \parallel \{a_1 = M_1\ ;\ \ldots a_n = M_n\} \equiv \{\ldots \{r \text{ with } a_1 = M_1\} \ldots \text{ with } a_n = M_n\}.$$

This equivalence could also have been deduced from the decomposition of s into one-field concatenations

$$(\ldots (r \parallel \{a_1 = M_1\}) \ldots \parallel \{a_n = M_n\}),$$

which is also the composition

$$(\{a_n = M_n\}^{\dagger} \circ \ldots \{a_1 = M_1\}^{\dagger})\ r.$$

We write

$$\{a_1 = M_1\ ;\ \ldots a_n = M_n\}^{\dagger}$$

[1] This is similar to the correspondence between *append* and *cons* on A-lists, in this particular case, the equality is $[M]$ *append* $r = M$ *cons* r.

for the abstraction of the previous expression over r. More generally we define the transformation $\dagger$ on record expressions, called *record abstraction*, by:

$$\begin{aligned}\{\}^\dagger &\equiv \lambda\, u.\ u\\ \{a = M\}^\dagger &\equiv \lambda\, u.\ \{u \text{ with } a = M^\dagger\}\\ (M \parallel N)^\dagger &\equiv N^\dagger \circ M^\dagger\end{aligned}$$

Since any record expression can be decomposed into a combination of the three previous forms, the transformation is defined for all records. It satisfies the property

$$r^\dagger = \lambda\, u.\ (u \parallel r).$$

Thus r is equal to $r^\dagger\ \{\}$. If we transform all record expressions in a program, then we have to replace the access $r.a$ by $(r^\dagger\ \{\}).a$. Actually, it is enough to apply r to a record r' that does not contain the a field and read the a field from the result $(r\ r').a$. In a typed language this solution will leave more flexibility for the type of r. Other constructs of the language simply propagate the translation. Thus the translation is completed by

$$\begin{aligned}(M.a)^\dagger &\equiv (M^\dagger\{\}).a\\ (\lambda\, x.\ M)^\dagger &\equiv \lambda\, x.\ M^\dagger\\ (M\ N)^\dagger &\equiv M^\dagger\ N^\dagger\\ x^\dagger &\equiv x\end{aligned}$$

The translation works quite well in an untyped framework. However, the encoding is not injective, for instance it identifies the empty record with the identity function. In the next section we adapt the translation to a typed framework.

4.1.2 The tagged translation

In this section we improve the translation so that the encoding becomes injective. The main motivation is to prepare the use of the encoding to get a typed version of $L^{\parallel}$ by pulling back the typing rules of a typed version of L. The well typed programs of $L^{\parallel}$ will be the reverse image of the well typed programs of L. The translation should be injective on well typed programs. A solution is to tag the encoding of records, so that they become tagged abstractions, distinct from other abstractions.

In fact we replace L by $L^{\mathrm{Tag},\mathrm{Untag}}$, that is L plus two constants Tag and Untag used to tag and untag values. The only reduction involving Tag or Untag is that Untag (Tag M) reduces to M. Tag and Untag can be thought as the unique constructor and the unique destructor of an abstract data type, respectively. In SML [48] they could be defined as:

```
abstype (α, β) tagged = Tagged of α → β with
  val Tag = fn x ⇒ Tagged x
  val Untag = fn Tagged x ⇒ x
end;
```

Their role is to certify that some functional values are in fact record abstractions, Tag stamps them and Untag reads and removes the stamps. Obviously, these constants are not accessible in $L^{\parallel}$, i.e. they are introduced during the translation only.

Syntactically the existence of Tag and Untag is not a question, but semantically a model of a calculus with record extension might not possess such constants. On the opposite, finding a

particular model in which the constants Tag and Untag exist might be as difficult as finding a direct model for concatenation. Anyhow, we limit our use of the encoding to syntactic issues.

The tagged translation is:

$$\begin{aligned}
\{\}^\dagger &\equiv \mathrm{Tag}\,(\lambda\, u.\ u)\\
\{a = M\}^\dagger &\equiv \mathrm{Tag}\,(\lambda\, u.\ \{u \text{ with } a = M^\dagger\})\\
(M \parallel N)^\dagger &\equiv \mathrm{Tag}\,(\lambda\, u.\ \mathrm{Untag}\ (N^\dagger\ (\mathrm{Untag}\, M^\dagger\ u)))\\
(M.a)^\dagger &\equiv ((\mathrm{Untag}\ M)\ \{\}).a
\end{aligned}$$

It does not modify other constructs:

$$\begin{aligned}
(\lambda\, x.\ M)^\dagger &\equiv \lambda\, x.\ M^\dagger\\
(M\ N)^\dagger &\equiv M^\dagger\ N^\dagger\\
x^\dagger &\equiv x
\end{aligned}$$

We would like to show a property such as: starting with a calculus of record extension, we can translate any program of a calculus with record concatenation into the first calculus enriched with constants Tag and Untag using the translation above, and thereby get — in some sense — an equivalent program.

$$L^{\parallel} \xrightarrow{\ \dagger\ } L^{\{\mathrm{Tag},\mathrm{Untag}\}}$$

$$\begin{array}{ccc}
M^{\parallel} & \longmapsto & M\\
\Big\downarrow{\scriptstyle \mathrm{eval}} & & \Big\downarrow{\scriptstyle \mathrm{eval}}\\
v^{\parallel} & \overset{?}{\cdots\cdots\cdots} & v
\end{array}$$

Without any such result, the translation † is no more than a good intuition to understanding record concatenation. In the next section it helps finding a type system for a language with concatenation $L^{\parallel}$ from a typed language with extension L, by translating $L^{\parallel}$ programs and then typing them in L.

4.1.3 Concatenation with removal of fields

We omitted one construction in the language L: the restriction of fields. We extend both languages L and $L^{\parallel}$ with record *restriction*:

$$M ::= \ldots \mid M \setminus a$$

Record restriction takes a record and removes the corresponding field from its domain. As for extension of fields, restriction of fields can be free or strict. We consider free restriction here. The question is obviously the extension of the transformation † to restriction of fields.

The guide line is to keep the equality

$$(M \setminus a)^\dagger = \lambda\, u.\ u \parallel (M \setminus a)$$

true, since it was true before the introduction of restriction of fields. Actually this equality is needed since it is the basis of the translation of the extraction of fields.

Unfortunately, the attempt

$$(M \setminus a)^\dagger = \lambda u.\ (M^\dagger u) \setminus a$$

does not work: the record $u \parallel (M \setminus a)$ is not equal to $(M^\dagger\ u) \setminus a$, since if the record u provides an a field, this field is defined in the left expression but it is undefined in the right expression. In fact $u \parallel (M \setminus a)$ is equal to $M^\dagger\ u$ on all fields but a. On the a field it is undefined if u is, or defined with the value of the a field of u otherwise. This operation cannot be written in the language L; we need another construct,

$$\{M \text{ but } a \text{ from } N\},$$

called *combining*. From two records M and N, it defines one that behaves exactly as M on all fields but a, and as N on the a field. This primitive is stronger than $(\_ \setminus a)$ which could be defined as $\{\_ \text{ but } a \text{ from } \{\}\}$.

Now, the translation of $(M \setminus a)$ can be defined by

$$(M \setminus a)^\dagger \equiv \lambda u.\ \{M^\dagger\ u \text{ but } a \text{ from } u\}$$

Its tagged version is:

$$(M \setminus a)^\dagger \equiv \mathrm{Tag}\,(\lambda\,.\ \{\mathrm{Untag}\,(M^\dagger\ u) \text{ but } a \text{ from } u\})$$

We call L^+ the language L extended with the combining construct. This construct has never been introduced in the literature before. If the language L is typed, it may be the case that the combining primitive cannot be assigned a correct and decent type in the type system of L and L^+ might not be a trivial extension of L or even not exist.

The combining construct is not in $L^{\parallel}$ and there is no easy way to provide it in an extension of $L^{\parallel}$. Therefore L but not L^+ is a sub-language of $L^{\parallel}$.

4.2 Application to a natural extension of ML

In this section we apply the translation where L is a version of ML with record extension, and we get a language with record concatenation. We first review the language Π taken from [100, 106] for record extension. Then we describe in detail two versions of the typed language $\Pi^{\parallel}$ obtained by pulling back the typing rules of Π. Last, we discuss the system $\Pi^{\parallel}$ on its own, and compare it with other existing systems with concatenation.

4.2.1 An extension of ML for records

The language, called Π, is taken from [100, 106]. It is an extension of ML, where the language of types has been enriched with record types in such a way that record operations can be introduced as primitive functions rather than built in constructs. The main properties are described in [106] and proved in [100, 104, 102]. The following summary should be sufficient for understanding the next sections. The reader is referred to [106] for a more thorough presentation.

Let $\mathcal{L}$ be a finite set of labels. We write a, b and c for labels and L for finite subsets of labels. The language of types is informally described by the following grammar (a formal description using sorts can be found in [106]):

$\tau ::= \alpha \mid \tau \to \tau \mid \Pi\left(\rho^{\emptyset}\right)$		types
$\rho^L ::= \chi^L \mid \mathrm{abs}^{\,L} \mid a : \varphi\ ;\ \rho^{L \cup \{a\}}$	$a \notin L$	rows defining all labels but those in L
$\varphi ::= \theta \mid \mathrm{abs} \mid \mathrm{pre}\,(\tau)$		fields

where α, β, γ and δ are type variables, χ, π and ξ are row variables and θ and ε are field variables.

Intuitively, a row with superscript L describes all fields but those in L, and tells for each of them whether it is present with a value of type τ (positive information $\mathrm{pre}\,(\tau)$) or absent (negative information $\mathrm{abs}\,$). A *template* row is either $\mathrm{abs}\,$ or a row variable. It always describes an infinite set of fields. The superscripts in row expressions L are finite sets of labels. Their main role is to prevent fields from being defined twice: the type

$$\Pi\left(a : \theta \ ; \ (a : \varepsilon \ ; \ \chi^L)\right)$$

cannot be written for any L. Similarly, all occurrences of the same row variable should be preceded by the same set of labels (possibly in a different order). The type

$$\Pi\left(a : \theta \ ; \ \chi^L\right) \to \Pi\left(\chi^L\right).$$

cannot be written either, since the row variable χ cannot be both in the syntactic class of rows not defining label a and the syntactic class of rows defining all labels. The superscripts are part of the syntax, but we shall omit them whenever they are obvious from context. We write $a : \alpha \ ; \ b : \beta \ ; \ \gamma$ for $a : \alpha \ ; \ (b : \beta \ ; \ \gamma)$.

Example 5 The following is a well-formed type:

$$\alpha \to \Pi\,(a : \mathrm{pre}\,(\alpha) \ ; \ b : \mathrm{pre}\,(\mathrm{num}) \ ; \ \mathrm{abs}\,)$$

Types are equal modulo the following equations:

- left commutativity, to reorder fields:

$$(a : \theta \ ; \ b : \varepsilon \ ; \ \chi) = (b : \varepsilon \ ; \ a : \theta \ ; \ \chi)$$

- distributivity, to access absent fields:

$$\mathrm{abs}\, = (a : \mathrm{abs}\, \ ; \ \mathrm{abs}\,)$$

Example 6 The record types $\Pi\,(a : \mathrm{pre}\,(\alpha) \ ; \ \mathrm{abs}\,)$ and $\Pi\,(b : \mathrm{abs}\, \ ; \ a : \mathrm{pre}\,(\alpha) \ ; \ \mathrm{abs}\,)$ are equal.

Any field defined by a template can be extracted from it using substitution if the template is a variable or distributivity if it is $\mathrm{abs}\,$.

Example 7 In $\Pi\,(a : \mathrm{pre}\,(\alpha) \ ; \ \mathrm{abs}\,)$, the template is $\mathrm{abs}\,$; its superscript is $\{a\}$. To read the b field, we replace $\mathrm{abs}\,$ by $(b : \mathrm{abs}\, \ ; \ \mathrm{abs}\,)$. The original type becomes $\Pi\,(a : \mathrm{pre}\,(\alpha) \ ; \ b : \mathrm{abs}\, \ ; \ \mathrm{abs}\,)$, and the new template has superscript $\{a, b\}$.

In $\Pi\,(a : \mathrm{pre}\,(\alpha) \ ; \ \chi)$, the χ variable can be substituted by $b : \varepsilon \ ; \ \pi$. The type becomes

$$\Pi\,(a : \mathrm{pre}\,(\alpha) \ ; \ b : \varepsilon \ ; \ \pi)$$

and π is the new template.

The language of expressions is the core ML language.

$$M ::= x \mid c \mid \lambda x.\, M \mid M\ M \mid \text{let } x = M \text{ in } M$$

where the constants c include the following primitives operating on records, with their types:

$$\begin{gathered}
\{\} : \Pi\,(\text{abs}\,) \\
_.a : \Pi\,(a : \text{pre}\,(\alpha)\ ;\ \chi) \ \to \alpha \\
\{_ \text{ with } a = _\} : \Pi\,(a : \text{abs}\ ;\ \chi) \ \to \alpha \to \Pi\,(a : \text{pre}\,(\alpha)\ ;\ \chi) \\
_ \setminus a : \Pi\,(a : \theta\ ;\ \chi) \ \to \Pi\,(a : \text{abs}\ ;\ \chi)
\end{gathered}$$

Primitives for Record Extension (Π)

The extension on a field $\{\_ \text{ with } a = \_\}$ is strict: a field can only be added to a record r that does not already possess this field. But the restriction of a field $\_ \setminus a$ is free: it can be applied to a record which does not have field a. Free extension with a field b is achieved by restriction of field b followed by strict extension with field b. That is, it is the composition:

$$(\_ \setminus a) \circ (\{\_ \text{ with } a = \_\}) : \Pi\,(a : \theta\ ;\ \chi) \ \to \alpha \to \Pi\,(a : \text{pre}\,(\alpha)\ ;\ \chi)$$

that we abbreviate $\{\_ \text{ with } !a = \_\}$. In the simplest language, the restriction of fields would not be provided, and the extension would be given whether strict or free.

Typing rules are the same as those of ML but where type equality is taken modulo the equations. As in ML, any typeable expression possesses a principal type. We show a few examples extracted from [106] and run on a CAML prototype.

Records are built all at once as in

```
#let car = {name = "Toyota"; age = "old"; registration = 7866};;
car : Pi (name : pre (string); registration : pre (num); age : pre (string); abs)
```

or from previous records by removing or adding fields:

```
#let truck = {car \ age with name = "Blazer"; registration = 6587867567};;
truck : Pi (name : pre (string); registration : pre (num); age : abs; abs)
```

Fields are accessed as usual with the "dot" operation.

```
#let registration x = x.registration;;
registration : Pi (registration : pre (α); χ) → α
```

Here, the field registration must be defined with a value of type α, so the field registration has type pre (α), and other fields may or may not be defined; they are grouped in the template variable χ. The return value has type α. The function eq below takes two records possessing at least a registration field of the same type[2]:

```
#let eq x y = equal (registration x) (registration y);;
eq : Pi (registration : pre (α); χ) → Pi (registration : pre (α); π) → bool

#eq car truck;;
it : bool
```

The identifier "it" is bound to the last toplevel phrase (the prototype types the expressions but it does not evaluate them). The two records car and truck do not have the same set of fields, but both can still be passed to the function registration.

[2]For simplicity of examples we assume the existence a polymorphic equality equal.

4.2.2 An extension of ML with record concatenation

The language Π described in section 4.2.1 can easily be extended with a combining primitive

$$\{\_ \text{ but } a \text{ from } \_\} : \Pi\,(a : \theta\ ;\ \chi) \ \to \Pi\,(a : \varepsilon\ ;\ \pi) \ \to \Pi\,(a : \varepsilon\ ;\ \chi)$$

The extended language is referred to as Π^+. We apply the transformation $\dagger$ with Π as L. We first consider the strict version of Π^+, then we show a few examples and we treat the free version of Π^+ at the end.

Symmetric concatenation

We encode the language $\Pi^{\|}$ with symmetric concatenation into the version of Π^+ with strict extension. We introduce a new type symbol $\{\_ \Rightarrow \_\}$ of arity two, and we assume given the two constants:

$$\begin{aligned} \text{Tag} &: (\Pi\,(\chi) \ \to \Pi\,(\pi)\,) \to \{\chi \Rightarrow \pi\}, \\ \text{Untag} &: \{\chi \Rightarrow \pi\} \to (\Pi\,(\chi) \ \to \Pi\,(\pi)\,). \end{aligned}$$

They are private to the translation.

A program is typable in $\Pi^{\|}$ if and only if its translation is typable in $\Pi^{\text{Tag},\text{Untag}}$ (Π extended with Tag and Untag). However, composing the translation with typechecking in $\Pi^{\text{Tag},\text{Untag}}$ is the same as typechecking in $\Pi^{\|}$ with the following types for primitives:

$$\begin{aligned} \{\} &: \{\chi \Rightarrow \chi\} \\ \_.a &: \{a : \text{abs}\ ;\ \chi \Rightarrow a : \text{pre}\,(\alpha)\ ;\ \pi\} \to \alpha \\ \{a = \_\} &: \alpha \to \{a : \text{abs}\ ;\ \chi \Rightarrow a : \text{pre}\,(\alpha)\ ;\ \chi\} \\ \backslash a &: \{a : \theta\ ;\ \chi \Rightarrow a : \varepsilon\ ;\ \pi\} \to \{a : \theta'\ ;\ \chi \Rightarrow a : \theta'\ ;\ \pi\} \\ \| &: \{\chi \Rightarrow \pi\} \to \{\pi \Rightarrow \xi\} \to \{\chi \Rightarrow \xi\} \end{aligned}$$

Primitives for symmetric concatenation ($\Pi^{\|}$)

Thus the translation can be avoided.

When typing directly in $\Pi^{\|}$ with the rules above, all record types are written with $\{\_ \Rightarrow \_\}$ and the type symbol Π can be removed; the grammar for types becomes

$$\tau ::= \alpha \mid \tau \to \tau \mid \{\rho^{\emptyset} \Rightarrow \rho^{\emptyset}\}$$

The type $\{\chi \Rightarrow \pi\}$ should be read "I am a record which given any input row of fields χ returns the output row π." The types for the primitives above can be read with the following intuition:

- The empty record returns the input row unchanged.
- As remarked above (section 4.1), we encoded the extraction of field a in M as the extraction of field a in the application of M to any record that does not contain the a field. Otherwise we would have got the weaker type:

  $$\_.a : \{\text{abs} \Rightarrow a : \text{pre}\,(\alpha)\ ;\ \chi\} \to \alpha$$

  Thus, the extraction of the a field of r takes a record r which, given any row where a is absent, produces a row where a is defined with some value v. The result $r.a$ is this value v.
- A one-field record extends the input row, defining one more field (that should not be previously defined).

- The removal of field a from a record M returns a record that acts as M except on the field a where it acts as the empty record.
- Finally, concatenation composes its arguments.

It is easy to see that any program in Π is also a program in $\Pi^{||}$. First, define the extension primitive by:

$$\{M \text{ with } a = N\} \equiv M \parallel \{a = N\}$$

It has type:

$$\{\chi \Rightarrow a : \text{abs} \ ; \ \pi\} \rightarrow \alpha \rightarrow \{\chi \Rightarrow a : \text{pre}\,(\alpha) \ ; \ \pi\}$$

Check that all the following typing assertions are correct in $\Pi^{||}$:

$$\begin{aligned} \{\} &: \{\text{abs} \Rightarrow \text{abs}\,\} \\ \_.a &: \{\text{abs} \Rightarrow a : \text{pre}\,(\alpha) \ ; \ \pi\} \rightarrow \alpha \\ \{\_ \text{ with } a = \_\} &: \{\text{abs} \Rightarrow a : \text{abs} \ ; \ \pi\} \rightarrow \alpha \rightarrow \{\text{abs} \Rightarrow a : \text{pre}\,(\alpha) \ ; \ \pi\} \\ \_ \setminus a &: \{\text{abs} \Rightarrow a : \varepsilon \ ; \ \pi\} \rightarrow \{\text{abs} \Rightarrow a : \text{abs} \ ; \ \pi\} \end{aligned}$$

Last, abbreviate $(\text{abs} \Rightarrow \rho)$ as (ρ) to conclude that $\Pi^{||}$ possesses all the primitives of Π with all types that Π can assign to them. The rest of the language Π is core ML and is also in $\Pi^{||}$.

Examples

We show a few examples processed by a prototype written in CAML [33, 122]. The type inference engine is exactly the one of Π; only the primitives have changed. The syntax is similar to CAML syntax.

The type of a one-field record says that the record cannot be merged with another record that also defines this field:

```
#let a = {a = 1};;
a : {a : abs; χ ⇒ a : pre (num); χ}
```

Two records r and s can be merged if they do not define common fields. For instance, r can be merged on the left with $\{a = 1\}$ if its output row on a is absent.

```
#let left r = r || {a = 1};;
left : {χ ⇒ a : abs; π} → {χ ⇒ a : pre (num); π}
```

The resulting record modifies its input row as r but on field a which is added. Similarly, s can be merged on the right with a if the input field a is present (with the adequate type).

```
#let right s = {a = 1} || s;;
right : {a : pre (num); χ ⇒ π} → {a : abs; χ ⇒ π}
```

In particular, s cannot define an a field, otherwise its input field a would be absent.

Non overwriting of fields is guaranteed on the left by negative information (absent field) at a positive row occurrence, and on the right by positive information (present field) at a positive row occurrence. Some symmetry is preserved! However writing r || s instead of s || r in a program sometime matters: one might typecheck while the other does not, though none of the programs would overwrite fields. If both typecheck, the type of the result will be the same (provided all fields are symmetric).

Here are a few more examples:

#let foo = fun r s → (r || s).a;;
foo : {a : abs; χ ⇒ π} → {π ⇒ a : pre (α); ξ} → α

This shows the functionality of concatenation on both sides. The result shall have an a field, but what argument will provide it is not specified yet.

#let gee = foo {b = 1};;
gee : {b : pre (num); a : abs; χ ⇒ a : pre (α); π} → α

Now r must define the a field.

#gee a;;
it : num

Asymmetric concatenation

The system Π may also provide free extension, with the following primitive:

$$\{\_ \text{ with } !a = \_\} : \Pi\,(a : \theta\ ;\ \chi) \ \to \alpha \to \Pi\,(a : \mathrm{pre}\,(\alpha)\ ;\ \chi)$$

This will make concatenation asymmetric:

$$\{!a = \_\} : \alpha \to \{a : \theta\ ;\ \chi \Rightarrow a : \mathrm{pre}\,(\alpha)\ ;\ \chi\}$$

For instance, the following example is typeable:

#let ab = (fun r → {!a = 1} || r) {!a = true; !b = 1};;
ab : {a : χ; b : π; ξ ⇒ a : pre (bool); b : pre (num); ξ}

This shows that asymmetric fields can be redefined with values of possibly incompatible types.

The choice between strict and free extension is encoded in the extension primitive, but the choice between asymmetric and symmetric concatenation is not encoded in the concatenation primitive which is always the composition. It is not concatenation which is symmetric or not, but record fields themselves! We can have symmetric and asymmetric fields coexisting peacefully.

#{!a = 1; b = true};;
it : {b : abs; a : χ; π ⇒ a : pre (num); b : pre (bool); π}

Primitives to modify these properties of fields can easily be provided

$$\begin{aligned}\mathrm{symmetric}^a &: \{a : \theta\ ;\ \chi \Rightarrow a : \mathrm{pre}\,(\alpha)\ ;\ \chi\} \to \{a : \mathrm{abs}\ ;\ \chi \Rightarrow a : \mathrm{pre}\,(\alpha)\ ;\ \chi\}\\ \mathrm{asymmetric}^a &: \{a : \theta\ ;\ \chi \Rightarrow a : \mathrm{pre}\,(\alpha)\ ;\ \chi\} \to \{a : \varepsilon\ ;\ \chi \Rightarrow a : \mathrm{pre}\,(\alpha)\ ;\ \chi\}\end{aligned}$$

But it is not possible to make all fields of a record symmetric, or asymmetric; this has to be done field by field.

We can now better understand why symmetric concatenation is not so symmetric. Both left and right functions accept any argument, and one should not expect them to behave the same on a record of which some of the fields are asymmetric.

With asymmetric fields, the following examples reach the limit of ML polymorphism. For instance, the function

#fun r s → s.b, r || s;;
it : {χ ⇒ b : abs; π} → {b : abs; π ⇒ b : pre (α); ξ} → α * {χ ⇒ b : pre (α); ξ}

does not accept a record r which has a b field, though the program would still run correctly if the b field of s is asymmetric. This is due to ML polymorphism weakness: the second argument is λ-bound and thus it is not polymorphic. The field b of s is observed by setting its input to abs, which has to be the output field b of r in r || s.

Since s has definitely a b field, the concatenation r || s is equal to the concatenation r\b || s. We can rewrite the previous program as

```
#fun r s → s.b, (r\b || s);;
it : {b : χ; π ⇒ b : ξ; 's} → {b : abs; 's ⇒ b : pre (α); 't} →
     α * {b : abs; π ⇒ b : pre (α); 't}
```

which can now be applied to any record r.

The restriction _\b of field b only changes the type of its arguments but does not modify it; it is called a *retyping function*. Many weaknesses of $\Pi^{||}$ originating in the restricted polymorphism provided by the ML type system can be solved by adding retyping functions. They insert type information in the program helping the type inference engine. We will describe other ways of solving these examples by strengthening the type inference engine in section 4.2.3.

4.2.3 Strength and weakness of $\Pi^{||}$

We compare our language with Wand's proposal [121], and Harper and Pierce's system and mention possible extensions.

Comparison with other systems

There are only a few other systems that implement concatenation. Wand's proposal [121] is still more powerful than our system $\Pi^{||}$. For instance

$$\lambda r.\ r.a + (\{a = 1\} \parallel r).a$$

is typable in Wand's system but not in ours. Wand's system polymorphism is carried by the concatenation operator, at the cost of bringing in the type system a restricted form of conjunctive types and having disjunction of principal types instead of unique principal types. In contrast, in our system, polymorphism is carried by records themselves. As mentioned above, we can regenerate polymorphism of records by inserting retyping functions. If the same restricted form of conjunctive types was brought in our system, then retyping functions would be powerful enough to regenerate all fields of a record without having to mention them explicitly. This would give back all the power of Wand's system.

This shows that the additional power of Wand's system comes from conjunctive types. Conversely, our system succeeds with only generic polymorphism on examples that needed conjunctive types in Wand's system. We are going to explain how this happens.

Wand's system can be reformulated in system Π. A simple idea is to type the concatenation operator by introducing an infix type operator $\parallel$ of arity two. Then concatenation has type:

$$\parallel\ : \Pi(\chi) \rightarrow \Pi(\pi) \rightarrow \Pi(\chi \parallel \pi)$$

But we have to eliminate $\parallel$ operators that might hide type collisions. In the system Π, we entice distributing concatenation on fields with the equations:

$$(a : \theta\ ;\ \chi) \parallel (a : \varepsilon\ ;\ \pi) = (a : \theta \parallel \varepsilon\ ;\ \chi \parallel \pi)$$

The operator $\|$ on fields can be defined by enumerating the triples $(\theta, \varepsilon, \theta \parallel \varepsilon)$. They are all triples of the form

$$(\theta, \mathrm{abs}, \theta) \quad \text{or} \quad (\theta, \mathrm{pre}(\beta), \mathrm{pre}(\beta)).$$

This disjunction in the relation $\|$ breaks the principal type property of type inference. Worse, disjunctions on different fields combine and make the resulting type (conjunction of types) explode in size.

Our system emphasizes that $\theta \parallel \varepsilon$ is uniform on θ: once we know ε, we can eliminate the conjunction in $\theta \parallel \varepsilon$. A field a, instead of carrying its type ε, carries the function $\theta \Rightarrow \theta \parallel \varepsilon$. For instance, if M has type τ, the record $\{a = M\}$ would have type $\Pi\,(a : \mathrm{pre}(\tau)\ ;\ \mathrm{abs})$ in Π. On field a, since ε is now $\mathrm{pre}(\tau)$, the merging $\theta \parallel \varepsilon$ is equal to $\mathrm{pre}(\tau)$. In the template, π is abs, and thus $\chi \parallel \pi$ is χ. We deduce the type of $\{a = M\}$ in $\Pi^{\|}$:

$$\{a : (\theta \Rightarrow \mathrm{pre}(\tau))\ ;\ (\chi \Rightarrow \chi)\} \qquad \text{i.e.} \qquad \{(a : \theta\ ;\ \chi) \Rightarrow (a : \mathrm{pre}(\tau)\ ;\ \chi)\}.$$

Another system with type inference was proposed by Ohori and Buneman in [86]. Their concatenation on records is recursive concatenation, which we do not provide. Note that they have a very restricted form of recursive concatenation since types in record fields must not contain any function type.

In explicitly typed languages, the only system with concatenation is the one of Harper and Pierce [50]; it implements symmetric concatenation. Since their system is explicitly (higher order) typed, we say that typing a $\Pi^{\|}$ program M succeeds in HP90 if we can find a HP90 program whose erasure (the program obtained by erasing all type information) is M. Their system has not free restriction of fields, but we shall ignore this difference.

The following $\Pi^{\|}$ program cannot be typed in HP90:

```
#let either r s = (r || s).a in
#  if true then either {a = 1} {b = 2} else either {b = 2} {a = 1};;
it : num
```

In the expression (r || s).a, one has to choose whether r or s is defining field a, and thus the function either cannot be used with the two alternatives. This breaks the symmetry of concatenation.

Conversely, there are programs that can be typed in HP90 but not in $\Pi^{\|}$ as a result of ML polymorphism restrictions. For instance the function

```
#let reverse r s = if true then r || s else s || r;;
reverse : {χ ⇒ χ} → {χ ⇒ χ} → {χ ⇒ χ}
```

cannot be applied to {a = 1} and {b = 2} in $\Pi^{\|}$. In HP90 it would have type

$$\forall\, \chi \cdot \forall\, \pi \# \chi \cdot \chi \to \pi \to (\chi \parallel \pi)$$

and could be applied to any two compatible records. It is difficult, though, to tell whether the failure comes from a limitation of polymorphism in general, or the inability to quantify with constraints, since the two are strongly related. The typability of the previous example in $\Pi^{\|}$ is somehow equivalent to the typability, in core ML, of the function:

```
#let reverse r s = if true then r o s else s o r;;
reverse : (α → α) → (α → α) → α → α
```

This is too weak a type! Whether a higher order language would give it a much better type is not so obvious. Next section provides a better basis for comparison between the two systems.

Limitations and extensions

Since the type inference engine of $\Pi^{\|}$ is the same as the one of Π (only types of primitives have changed), both systems enjoy the same properties. Record polymorphism is provided by ML genericity introduced in let bindings. If this is too restrictive, then one should introduce type inclusion. One could also have a restricted conjunctive engine as in [121]; however this would decrease considerably the efficiency of type inference, and the readability of types. Allowing recursion on types would also require an extension of the results (though in practice the mechanism is already present). In $\Pi^{\|}$, as in Π, present fields cannot be implicitly forgotten, but have to be explicitly removed, unless the structure of fields is enriched with flags. All these improvements are discussed in detail in [106].

4.3 Other applications

The transformation can also be applied to other languages, which we illustrate in this section.

4.3.1 Application to Harper and Pierce's calculus.

The higher order typed language of Harper and Pierce [49] already possesses concatenation, but records are not abstractions. It can still benefit from the encoding. Instead of presenting special constructs for operations on records, we could assume given the following primitives in their language:

$$\begin{aligned}
\{\} &: \Pi\,() \\
_.a &: \forall\,\alpha \cdot \forall\,\chi \# a \cdot (\Pi\,(a:\alpha) \;\|\; \chi) \to \alpha \\
\{a = _\} &: \forall\,\alpha \cdot \alpha \to \Pi\,(a:\alpha) \\
\backslash a &: \forall\,\alpha \cdot \forall\,\chi \# a \cdot (\Pi\,(a:\alpha) \;\|\; \chi) \to \chi \\
\| &: \forall\,\chi \cdot \forall\,\pi \# \chi \cdot \chi \to \pi \to (\chi \,\|\, \pi)
\end{aligned}$$

Primitives for HP90

But the type system is not enough sophisticated to type the primitive $\{\_$ but a from $\_\}$. Thus we apply the translation dropping the removal of fields. Using the encoding, the primitive operations on records in the language HP90$^{\|}$ have the following types:

$$\begin{aligned}
\{\} &: \forall\,\chi \cdot (\chi \Rightarrow \chi) \\
_.a &: \forall\,\alpha \cdot \forall\,\chi \# a \cdot \forall\,\pi \# a \cdot (\chi \Rightarrow \Pi\,(a:\alpha) \;\|\; \pi) \to \alpha \\
\{a = _\} &: \forall\,\alpha \cdot \alpha \to \forall\,\chi \# a \cdot (\chi \Rightarrow \Pi\,(a:\alpha) \;\|\; \chi) \\
\| &: \forall\,\chi \cdot \forall\,\pi \cdot \forall\,\xi \cdot (\chi \Rightarrow \pi) \to (\pi \Rightarrow \xi) \to (\chi \Rightarrow \xi)
\end{aligned}$$

Primitives for HP90$^{\|}$

We can define a function either:

$$\begin{aligned}
&\Lambda\,\alpha.\ \Lambda\,\chi.\ \Lambda\,\pi \# a.\ \lambda\,r : (\Pi\,() \;\Rightarrow \chi).\ \lambda\,s : (\chi \Rightarrow \Pi\,(a:\alpha) \;\|\; \pi). \\
&\quad (_.a\ [\alpha]\ [\Pi\,()\,]\ [\pi]\ \ (\|\ \ [\Pi\,()\,]\ [\chi]\ [\Pi\,(a:\alpha) \;\|\; \pi]\ \ r\ \ s))
\end{aligned}$$

and apply it to records $\{\mathsf{a} = 1\}$ and $\{\mathsf{b} = 2\}$ in any order. For instance,

$$\text{either}\ [\text{num}]\ [\Pi\,(a:\text{num})\,]\ [\Pi\,(b:\text{num})\,]\ \ (\{a=1\}\ [\Pi\,()\,])\ \ (\{b=2\}\ [\Pi\,(a:\text{num})\,])$$

This example is not typable in HP90.

Conversely, the program:

```
let reverse r s = if true then r || s else s || r in reverse {} {a = 1}, reverse {} {a = 1};;
```

can be typed in HP90, but we conjecture that it cannot be typed in HP90$^{\|}$. In fact its typability in HP90$^{\|}$ is equivalent to the following term being the erasure of a term of F:

```
(fun r → K (r I K) (r K I))
        (fun f g → (fun x → f (g x)) or (fun x → g (f x)))
```

where I is fun x → x and K is fun x y → x, and or is a constant assumed of type $\Lambda\,\alpha.\ \alpha \to \alpha \to \alpha$ in F.

To summarize, none of the language HP90 or HP90$^{\|}$ would be more powerful than the other. Remark that type applications and type abstractions are located at completely different places, thus a partial translation of explicitly typed terms from HP90$^{\|}$ to HP90 can only be global.

A previous language proposed by Harper and Pierce in [49] had no concatenation, but shared the same spirit as HP90. The transformation applies to it as well, and results in a language with concatenation very closed to HP90$^{\|}$.

4.3.2 Application to Cardelli and Mitchell's calculus.

Unlike HP90, the language of Cardelli and Mitchell [27] does not already provide concatenation of records, but only strict extension. The application of our encoding to CM89 is not harder than to HP90. The language cannot be easily extended with the combining construct, therefore we skip the removal of fields. Using CM89 types, primitives for record operations in CM89$^{\|}$ have the following types:

$$\begin{aligned}
\{\} &: \forall\,\chi \cdot (\chi \Rightarrow \chi)\\
_.a &: \forall\,\alpha \cdot \forall\,\chi < \langle\langle\rangle\rangle \setminus a \cdot \forall\,\pi < \langle\langle\rangle\rangle \setminus a \cdot (\chi \Rightarrow \langle\langle \pi \parallel a : \alpha\rangle\rangle) \to \alpha\\
\{a = _\} &: \forall\,\alpha \cdot \alpha \to \forall\,\chi < \langle\langle\rangle\rangle \setminus a \cdot (\chi \Rightarrow \langle\langle \chi \parallel a : \alpha\rangle\rangle)\\
\| &: \forall\,\chi \cdot \forall\,\pi \cdot \forall\,\xi \cdot (\chi \Rightarrow \pi) \to (\pi \Rightarrow \xi) \to (\chi \Rightarrow \xi)
\end{aligned}$$

Primitives for CM89$^{\|}$

We can again define the function either:

$$\begin{aligned}
&\Lambda\,\alpha.\ \Lambda\,\chi.\ \Lambda\,\pi < \langle\langle\rangle\rangle \setminus a.\ \lambda\,r : (\langle\langle\rangle\rangle \Rightarrow \chi).\ \lambda\,s : (\chi \Rightarrow \langle\langle \pi \parallel a : \alpha\rangle\rangle).\\
&\quad (_.a\ [\alpha]\ [\langle\langle\rangle\rangle]\ [\pi]\ \ (\|\ \ [\langle\langle\rangle\rangle]\ [\chi]\ [\langle\langle \pi \parallel a : \alpha\rangle\rangle]\ \ r\ \ s))
\end{aligned}$$

and apply it to the records $\{a = 1\}$ and $\{b = 2\}$:

$$\text{either}\ [\text{num}]\ [\langle\langle a : \text{num}\rangle\rangle]\ [\langle\langle b : \text{num}\rangle\rangle]\ \ (\{a = 1\}\ [\langle\langle\rangle\rangle])\ \ (\{b = 2\}\ [\langle\langle a : \text{num}\rangle\rangle])$$

4.3.3 Multiple inheritance without record concatenation

Multiple inheritance has been encoded with record concatenation [121]. We have encoded record concatenation with record extension. By composition, multiple inheritance can be encoded with record extension.

Given the strengthening of the type inference engine to recursive types, the system $\Pi^{\|}$ would support multiple inheritance as presented in [121]. But multiple inheritance makes very little use of concatenation. It is only necessary for building new methods, but objects do not need it. Thus it may be worth revisiting the typechecking of multiple inheritance of [121] and eliminating the need for concatenation by abstracting methods as we abstracted records.

The following encoding of multiple inheritance was used by Wand in [121]. The definition of a class

$$\text{class}\ (\vec{x})\ \text{inherits}\ \overrightarrow{P(\vec{Q})}\text{methods}\ \overrightarrow{a = M}\ \text{end}$$

was encoded as

$$\lambda\,\vec{x}.\ \lambda\,self.\ \overrightarrow{P(\vec{Q})}\ \|\ \{\overrightarrow{a = M}\}$$

The creation of objects of that class

$$\text{instance}\ C(\vec{N})$$

was the recursive expression

$$Y(C(\vec{N}))$$

Sending a method a to an object x was the same as reading the field x of a. The problem with this encoding is that it requires record concatenation. We can easily get read of it, using our trick. We encode a class definition as

$$\lambda\,\vec{x}.\ \lambda\,u.\ \lambda\,self.\ u\ \|\ \overrightarrow{P(\vec{Q})}\ \|\ \{\overrightarrow{a = M}\}$$

i.e.

$$\lambda\,\vec{x}.\ \lambda\,u.\ \lambda\,self.\ \{\overrightarrow{P(\vec{Q})}\ \circ\ u\ \text{with}\ \overrightarrow{a = M}\}$$

which only requires record extension. Then creating an object of that class becomes

$$Y(C(\vec{N})\{\})$$

and sending a method is unchanged.

Remarks

Since removing of fields is not needed here, this section applies to all typed calculi with record extension.

This section uses Wand's conception of inheritance. Objects are carrying their dictionaries. Other views of objects do not encode with record operations. This section does not apply to them.

Conclusion

We have described how a functional language with records and record extension automatically provides record concatenation. Though records are data, they should be typed as if there were abstractions over an input row of fields that they modify. Their behavior can be observed at any time by giving them the empty row as input. Concatenation is then composition.

We have applied the method to a record extension of ML. We have obtained a language implementing all operations on records except the recursive merge, allowing type inference in a very efficient way in practice.

The kind of type system that we have obtained seems complementary to Harper and Pierce's one. Taking the best of the two systems would be interesting investigation.

The encoding also helps understanding concatenation. However, the relationship between the semantics of a program in the language with concatenation and the semantics of its translation need to be investigated closely before claiming that concatenation itself comes for free.

Acknowledgments

I am grateful for interesting discussions with Luca Cardelli, Georges Gonthier, Jean-Jacques Lévy, Benjamin Pierce and Mitchell Wand, and particularly thankful to Xavier Leroy whose comments on the presentation of this article were very helpful.

Chapter 5

Programmer les Objets dans ML-ART Une extension de ML avec des types abtraits et des types enregistrements

Ce chapitre a été publié dans [108].

Programmer les objets dans ML-ART

Dans une approche avec classes, les objets peuvent être programmés directement et efficacement dans une extension simple de ML. Leur représentation, qui s'appuie sur des types enregistrements et des types abstraits, permet toutes les opérations usuelles telles que l'héritage multiple, la possibilité pour une méthode de retourner l'objet lui-même ou de lui envoyer un message, y compris un message sa classe parente. Toutefois, la coercion du type d'un objet vers son type correspondant dans une classe parente reste explicite. Nous donnons aussi une représentation plus simple où les objets ne sont plus des valeurs récursives. Le langage sous-jacent est une extension de ML avec des types récursifs, des types existentiels et universels et des enregistrements extensibles mutables. Le langage ML-ART est équipé d'une sémantique en appel par valeur pour laquelle la correction du typage est prouvée.

Programming objects in ML-ART

Class-based objects can be programmed directly and efficiently in a simple extension to ML. The representation of objects, based on abstract and record types, allows all usual operations such as multiple inheritance, object returning capability and message transmission to themselves as well as to their super classes. There is, however, no implicit coercion from objects to other objects of their super-classes. A simpler representation of objects without recursion on values is also described. The underlying language extends ML with recursive types, existential and universal types, and mutable extensible records. The language ML-ART is given with a call by value semantics for which type soundness is proved.

Introduction

An important motivator for type-checking extensible records is the records' application to object encoding. Initiated by Cardelli in 1984 [20], continued by Wand [119], and then many others, type-checking records has produced several satisfactory solutions for higher order languages [27, 49] and for ML [86, 84, 106]. Object encoding, based on record calculi, reveals severe difficulties, mainly due to overreliance on recursive values. Thus, the tendency has been to design languages with objects as primitive operations [15, 3, 51, 82], rather than encodings, to achieve important simplification of type-theoretical models.

Pierce et al. produces convincing evidence that object oriented programming could be treated as a matter of programming style, at least from a theoretical point of view [93]. However, the use of $F^{\omega}_{\subset}$ as the basis language supports the idea that encodings involve complex type theories, and the demonstration does not always apply to the ML programmer. The need to write many coercions, due to the use of explicit types and the absence of record extension, makes it obvious that large-scale object-oriented applications cannot be programmed directly in $F^{\Omega}_{\subset}$. Finally, the encoding is created in a call-by-name language, which results in a duplication of too many structures. A recent version of the encoding in a call-by-value language [92] still contains inherent inefficiencies. At least a large amount of syntactic sugar must be provided to program objects in $F^{\omega}_{\subset}$.

We concur with the claim that object-oriented programming is essentially a matter of style. Consequently, it is not addressed in this paper. Our main goal is to demonstrate that objects can be programmed in a small extension to ML. Therefore, we repeat Pierce's method using, instead, a basic language derived from ML. This results in a quite elegant and still flexible class-based object-oriented programming style, almost as concise as if objects were primitive. No syntactic sugar is required. This approach allows for programming capabilities such as multiple inheritance, object returning ability and message transmission to themselves as well as to their super classes. However, implicit coercion of objects to their counterparts in super classes is not implemented.

As in [93], we consider objects as abstract data structures, but our encoding differs in two essential ways. First, we can take advantage of record extension to implement inheritance in a simpler way that avoids successive coercions and treats classes as "first class citizens". Ignoring implicit class coercions enables to move the recursion on "self" from method vector creation to method application, converting objects to non recursive values.

Another interest of this paper is the language ML-ART utilized for programming objects; it extends core ML with several orthogonal features. None of these is really new but itself, however, the combination is original. We give a complete definition of ML-ART, omit type inference, but verify type soundness.

The most important feature of ML-ART is extensible records. We choose those described in [106], althoug other choices are permissible, provided they implement polymorphic access and polymorphic extension. Polymorphic access refers the ability to define a function that reads the same field of many records, with different domains. This is the key operation to messages passing. Record extension is the operation that creates new records from older ones by addition of new fields. It is said to be polymorphic if it can operate on records with different domains. Record extension is used to program single inheritance [119] or even multiple inheritance reusing the trick that provides record concatenation with only record extension [107]. Polymorphic record extension also allows polymorphic duplication of records, which is used to copy the state without knowing its representation.

The language is also enriched with recursive types. Record types and recursive types are sufficient to program objects with value abstraction, modulo serious inefficiencies and difficulties

with the compilation of recursive values. Thus, we choose to extend the language with existential types, as suggested by Laüfer and Odersky [64], and use type abstraction to conceal the internal state of objects. This necessitates replacement of record types with more expressive projective types [103]. Finally, existential types introduce scope borders that can only be crossed using universal types in a dual way.

The paper is organized as follows. In section 5.1, we briefly describe encoding of records using value-abstraction and illustrate various problems that could find only *ad hoc* solutions. In section 5.2 we informally introduce the language ML-ART. We motivate and describe the different features one by one. A formal presentation is supplied in the appendices. In section 5.3, we describe two object programming styles based on a number of variations of points. In the final section, we discuss and conclude the experience.

5.1 Failure to program objects with value abstraction

In this section we attempt to program objects with value abstraction until we meet serious difficulties.

All examples are run in a sub-language of ML-ART composed of ML plus extensible records and recursive types[1]. The key features that are used in this section are just polymorphic access and polymorphic extension. Any language that provides those two operations could be used instead. The paper is organized such that all language features are described in the next section; although the constructions used here are very simple and can be mostly understood intuitively, it is possible to read the three first parts of the next section before considering the following examples.

The language ML-ART is strongly typed and provides type inference. However, objects have anonymous, long, and often recursive types that describe all methods that the object can receive. Thus, we usually do not show the inferred types of programs in order to emphasize object and inheritance encodings rather than typechecking details. This is quite in the spirit of ML where type information is optional and is mainly used for documentation or in module interfaces. Except when trying top-level examples, or debugging, the user does not often wish to see the inferred types of his programs in a batch compiler. When printed, the output of compilation is indicated with a marginal "|" sign.

As in [93], we consider objects as pairs (R,M) of an internal state R and a method vector M. The state is the data stored in the object; it is usually different for each object and mutes when the object receives appropriate messages, which drives the behavior of objects on messages.

```
let pointR v = {!x = v};;
```

The method vector is a record of methods; when the object receives a message, the corresponding method is extracted from the record and is applied to the state of the object. Each method is thus a function whose first argument is always the state (even if the method does not uses the state, for uniformity reasons).

```
let pointM = let getx R = R.x  and setx R x = R.x ← x in {getx; setx};;
```

Point objects are created as follows:

```
let new_point v = (pointR v, pointM);;
```

As described above, sending the method getx to a point is realized by the function:

[1] An implementation of this sub-language as an extension of the Caml-Light language, should be released soon. The full ML-ART language is built on top of this extension and is still experimental.

```
let send_getx (R,M) = M.getx R;;
```

For example,

```
send_getx (new_point 1);;
- : int = 1
```

However, point objects expose their representation R to the entire world. This is a failure to guarantee the abstraction that is usually expected in programming with objects. Moreover it prevents from having several implementations of similar objects with the same interface (accepting the same messages) but different representations. For instance, points with polar and carthesian coordinates could not be mixed since their representations, and thus part of their types, would be incompatible.

Hiding the representation in the object $(\mathtt{R}, \vec{\mathtt{m}})$ can be accomplished by partially applying each method m to the internal state and representing the object by $\overrightarrow{\mathtt{m}(\mathtt{R})}$.

```
let pointM R = let getx () = R.x  and setx () x = R.x ← x in {getx; setx};;

let new_point v = pointM (pointR v);;
```

Sending a message accesses the right method and passes the argument () instead of the state to launch the method. For example,

```
let send_getx P = P.getx ();;

send_getx (new_point 1);;
```

There is already an important efficiency problem. Since pointM is abstracted on state R, each object will be created with a new method vector pointM R. Hiding by value abstraction is realized by re-arranging the program, fortunately keeping its high-level semantics. However, the operational behavior of the program or, equivalently, a low-level semantics that would count resources, has been seriously altered. In practice, the method vector may be quite large, and it is unrealistic —except maybe for school examples— to have a copy of the method vector in each object of the same class.

Ignoring efficiency issues and pursuing this exercise is still an interesting experiment. Notions of classes and inheritance could be implemented in a similar way to what is done below for objects with type abstraction. However, as in section 5.3.1 we would hit the same problem of creating recursive values in non trivial ways. We did not find any but *ad hoc* solution to both of these problems, yet.

Refusing to twist the language, we consider the value-abstraction approach to objects as a failure. Instead, taking the type-abstraction approach, the goal of this paper can be fulfilled after a few powerful language extensions. In the view of objects as pairs (R,M) of an internal state R and a method vector M, the type of R, say τ_R must be hidden, but still allow R to be passed to any field of M. Thus, it is necessary to remember that each method of M has type $\tau_R \to \tau_m$, whatever the type τ_R is. Methods of the same object will have, of course, independant codomains τ_m and different objects will answer different set of messages. Finding a type for objects means finding a uniform way of saying *This is a pair whose first component has some type τ_R and whose second component is a record of functions of common domain τ_R, and exhibiting their codomains.*

5.2 The language

The language ML-ART is designed by adding several features to core ML. Each extension is quite simple and none of them is really new; they have been either described somewhere else or already

implemented in some version of ML. Their combination provides just enough power to program objects in a very flexible and elegant way. The two main extensions are polymorphic records and existential types. Recursive types are also added and record types are enriched to projective types. Finally, it is convenient to have universal types and mutable fields. First, core ML is briefly described, then each of the feature is introduced independently before the full language ML-ART is presenetd.

5.2.1 The core language

The core language is ML, with a call by value semantics. Programs are given by the following grammar:

$$a ::= x \mid \texttt{fun}\ x \to a \mid a_1\ a_2 \mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2$$

and are taken modulo renaming of bound variables. The conditionals and pairs may be provided as syntactic sugar. For convenience, we also use simultaneous let bindings with the construction:

$$\texttt{let}\ x_1 = a_1 \text{ and } \ldots\ x_n = a_n\ \texttt{in}\ a_0$$

They can be expanded into cascades of lets after renaming of bound variables. Multiple-case data-types could easily be added together with pattern matching, but we will not need them in the examples. We do not include reference cells in the initial language either since they are subsumed by mutable record fields. For similar reason, we have no construction for recursion.

Types expressions are given by the grammar:

$\tau ::= \alpha \mid \tau \to \tau$	types
$\sigma ::= \tau \mid \forall\alpha \cdot \tau$	type schemes
$A ::= \emptyset \mid A, x : \sigma$	contexts

We abbreviate sequences of quantifiers $\forall\alpha_1 \cdot \ldots \forall\alpha_n \cdot \sigma$ by $\forall\alpha_1, \ldots \alpha_n \cdot \tau$ and often write $\vec{\alpha}$ for tuples of variables. We write $[\tau/\alpha]$ the substitution that replaces free occurrences of α by τ.

The Damas-Milner typing relation $A \vdash a : \sigma$ is defined in the appendix by the inference rules of figure 5.7.

5.2.2 Extensible records

Monomorphic records, as in Sml [48] or Caml-Light [71], are not sufficient to program objects. The basic operation on objects is message passing that is usually implemented as an access to the appropriate message in a vector of methods carried by the object itself. The same message often need to be passed to different classes of objects that can receive different sets of messages. Thus access to the method vector must be polymorphic.

Record extension is not absolutely required for simulating objects. For instance, in [93], classes are defined at top-level so that when a class inherits another all methods of the super class are known and can be explicitly copied into the new method vector. However, writing all coercion functions quickly becomes a burden and some syntactic sugar is required to automatically generate them. Non polymorphic record extension can be useful to avoid syntactic sugar, but classes cannot yet be first class citizens ([52]). Polymorphic extension make it possible to program multiple inheritance and to treat classes as first citizens.

The extensible records are those[2] presented in [106]. We assume given a denumerable collection of labels $\mathcal{L}$. Instead of introducing nex syntac or records, we extend the set of constants with the empty record $\{\}$ and two families of primitives $(\_.\ell)$ and $(\_ \parallel \{\ell = \_\})$ for all labels ℓ, implementing respectively the access to field ℓ and extension on field ℓ. For convenience, we also write $(a \parallel \{\ell_1 = a_1 \; ; \ldots \ell_n = a_n\})$ as a short hand for $(\ldots(a \parallel \{\ell_1 = a_1\})\ldots \| \{\ell_n = a_n\})$ and, as in Sml, the abusive but very convenient convention that $(a \parallel \{x\})$ stands for $(a \parallel \{\ell_x = x\})$, where ℓ_x is the label that has the same name as variable x.

The type system is enriched with record types:

$$\tau ::= \ldots \mid \{\tau\} \mid \tau.\tau \mid \texttt{abs} \mid \texttt{pre} \mid (\ell : \tau; \tau) \quad \ell \in \mathcal{L}$$

The formation of types is restricted by sorts. Type symbols `abs` and `pre` may only appear on the left hand sides of dots; they tell whether the corresponding field is accessible or not. See [106] or the appendice for a detailed treatment of sorts.

There is no special typing rules for records; the primitives simply come with the following principal types:

$$\begin{aligned} \{\} &: \forall\alpha \cdot \{\texttt{abs}.\alpha\} \\ (\_.l) &: \forall\alpha_0, \alpha_1 \cdot \{l\colon \texttt{pre}.\alpha_0 \; ; \alpha_1\} \to \alpha_0 \\ (\_ \parallel \{\ell = \_\}) &: \forall\alpha_0, \alpha_1, \alpha_2 \cdot \{l\colon \alpha_0 \; ; \alpha_1\} \to \alpha_2 \to \{l\colon \texttt{pre}.\alpha_2 \; ; \alpha_1\} \end{aligned}$$

Here is an example combining most of the constructs:

```
let get x = x.a in let ra = {a = 1} in let rb = ra || {a = 2; b = true} in
get ra + get rb;;
- : int = 3
```

5.2.3 Mutable data structures

ML reference cells could be added to the language, but the mutable data structures that have been introduced in Caml and re-used in Caml-Light [71] are more flexible and in fact more powerful. In those languages record data structures are dual of variant data structures, and need to be declared in the same way with all their labels. Labels carry implicit type information according to the last record definition in which they appear. Fields whose labels have been declared mutable can then be updated. For instance, it is possible to write the following program in Caml-Light:

```
type person = {name : int; !age : int};;
let birthday person = person.age ← person.age + 1;;
```

With declared records, mutable fields save space but do not increase power, since they could always be replaced by non mutable fields that contain references to the dynamic values.

Polymorphic extension may also be used to dupplicate records. For instance, the function copy

```
let copy r = r || {known = r.known};;
```

dupplicates all records having at least a known field. It is then natural to allow multiple extension of no fields and interpret it as copying its argument:

```
let copy r = r || {};;
```

[2] In fact, in [106] we presented two variants of record types, both described in section 3.3. Here, we use the second one, but with the weaker type assumptions of the first one.

Polymorphic duplication of records becomes an interesting feature when fields are mutable; it is used below to create new objects by copying, and then modifying, the state of an older object. Polymorphic duplication could not be written if fields were coded as references.

Mutability information is carried by labels, and must be mentioned explicitly at the construction. We replace record extension by two primitives: $(\_ \parallel \{\ell\colon \_\})$ and $(\_ \parallel \{!\ell\colon \_\})$ for non mutable and mutable field extensions. We also add a new primitive $(\_.\ell \leftarrow \_)$ for field mutation.

In order to carry mutability information in types, the `pre` type symbol becomes of arity 1, and we add two constant symbols `mut` and `static`. Sorts guarantee that only the type symbols `mut` and `static` may appear under `pre`. The grammar of types is updated to:

$$\tau ::= \ldots \mid \{\tau\} \mid \tau.\tau \mid \mathtt{abs} \mid \tau\,\mathtt{pre} \mid \mathtt{mut} \mid \mathtt{static} \mid (\ell : \tau; \tau) \qquad \ell \in \mathcal{L}$$

Record primitives are given below

$$\begin{aligned}
\{\} &: \forall\alpha \cdot \{\mathtt{abs}.\alpha\} \\
(_.l) &: \forall\alpha_0,\alpha_1,\alpha_2 \cdot \{\ell\colon \alpha_0\,\mathtt{pre}.\alpha_1\ ;\ \alpha_2\} \to \alpha_1 \\
(_.l \leftarrow _) &: \forall\alpha_0,\alpha_1,\alpha_2 \cdot \{\ell\colon \mathtt{mut}\,\mathtt{pre}.\alpha_1\ ;\ \alpha_2\} \to \alpha_1 \to \mathtt{unit} \\
(_ \parallel \{\ell = _\}) &: \forall\alpha_0,\alpha_1,\alpha_2 \cdot \{\ell\colon \alpha_0\ ;\ \alpha_1\} \to \alpha_2 \to \{\ell\colon \mathtt{static}\,\mathtt{pre}.\alpha_2\ ;\ \alpha_1\} \\
(_ \parallel \{!\ell = _\}) &: \forall\alpha_0,\alpha_1,\alpha_2 \cdot \{\ell\colon \alpha_0\ ;\ \alpha_1\} \to \alpha_2 \to \{\ell\colon \mathtt{mut}\,\mathtt{pre}.\alpha_2\ ;\ \alpha_1\}
\end{aligned}$$

In particular the access is polymorphic in mutability as well, which enables $(\_.\ell \leftarrow \_)$ to be applied to records with either mutable or static field ℓ. As an example, references are definable:

```
let newref x = {!val = x}
and assign r x = r.val ← x
and deref r = r.val;;
newref : β → {val : mut pre.β; abs.γ} = ⟨fun⟩
assign : {val : mut pre.β; γ} → β → unit = ⟨fun⟩
deref : {val : β pre.γ; δ} → γ = ⟨fun⟩
```

In order to guarantee the type-safety in the presence of mutable objects, we choose the "polymorphism on values" approach [124]. In fact, we refine this approach in order to get polymorphic record extension, although record extension is not a value. We define a set of generalizable terms ranged over by letter b that do not contain any application except applications of safe constants and applications under abstractions. The only dangerous constant is record extenmsion on a mutable field. The grammar of generalizable terms is given (for semantics that do not evaluate under abstractions) in table 5.1 of the appendix. Restricting the expression of GEN rule (figure 5.7) to be generalizable guarantees that the evaluation of generalizable terms never create any mutable data that is used polymorphically. For instance, let bound expression that are not generalizable expressions can only be assigned monomorphic types.

5.2.4 Recursive types

The notion of "self" allows an object to send messages to itself or, worse, to return itself (or a copy of itself) when it receives some appropriate message. This makes it possible to implement fix points without ML recursive definitions. Thus, recursive types are required. Since we wish not to declare the types of objects (although we will allow to do so for documentation), we need to infer recursive types.

Recursive types are only provided through data type declarations in ML. Allowing implicit recursive types would be quite easy [105] since type inference reduces to first order unification for

which there are well known algorithms in the presence of recursive types [77, 54]. This does not mean that implicit recursive types would make recursive type declarations obsolete. For instance, in ML-ART, there is no record type equivalent to the Caml Light type definition below, since such a type cannot be represented by a regular tree (but it can be regularly generated).

type α foo = {foo : (α * α) foo};;

Other interesting applications of implicit recursive types in ML can be read in [68].

To model recursive types we extend types with the syntactic construct

$$\tau ::= \ldots \mid \texttt{rec}\ \alpha.\tau$$

Equality for recursive types is defined in appendix 5.4.3.

The following fix-point combinator is definable with recursive types,

```
let Y  F = (fun f → f f) (fun f x → F (f f) x);;
Y : ((β → γ) → β → γ) → β → γ = ⟨fun⟩
```

Recursive definition of functions `let rec` $f = a_1$ `in` a_2 is allowed as syntactic sugar for `let` $f =$ `Y (fun` $f \rightarrow a_1$`) in` a_2. Recursively let-bound variables become lambda-bound in the expanded form, which provides the correct monomorphic typing rules for recursion.

The type of `Y` forces a_1 to have a functional type and does not allow the construction of recursive values. It is possible to have a primitive construct for recursion that permits some definitions of recursive values, but it is difficult to automatically filter admissible definitions of non-functional recursive values as soon as application is tolerated. In core ML-ART we forbid recursion on non-functional values. Still, fix point of non-functional values are used in some of the examples, but the reader will always be warned.

5.2.5 Projective types

We ended previous section claiming that a good type system for programming objects must provide a way of referring to records of functions with all the same domains, and defining their codomains. Simple record types do not allow to write any information in template types. Presently, a template of a record type can either be a field variable α, as in the type of the access primitive, or the expression `abs`.α, as in the type of the empty record.

It would be easy to allow type-like expressions inside templates, for instance $\alpha_0.\alpha_1 \rightarrow \alpha_2$, would force any field to be an arrow type. But variables α_1 and α_2 are template variables and can just serve as filters to types of fields. In order to constraint all fields to have the same domain α_0, type variable α_0 must be coerced to a template term. This is exactly what projective types allow. The grammar of types is extended with a new symbol `row _`:

$$\tau ::= \ldots \mid \texttt{row}\ \tau$$

Sorts allow `row _` to coerce an ordinary type expression to a template expression; sorts are relaxed such that all ordinary type symbols but {_} may occur inside templates. For instance, $\{\alpha_0.\texttt{row}\ \tau \rightarrow \alpha_1\}$ is the type of all records whose defined fields are functions of the same domain τ.

Projective types are described in more details in the appendix 5.4.2 and are fully formalized in [104] and also more intuitively described in [103]. They enjoy all the interesting properties of record types. Although projective types are richer that record types, ML-ART does not have new language constructs; the additional power is mainly used to write more expressive existential types.

5.2.6 Existential types

Existential types are the basic tool for defining objects with type abstraction. An extension of ML with existential types has been proposed by K. Läufer and M. Odersky in [64]. We slightly simplify the presentation of their proposal by separating existential types from variant types.

We first extend type schemes with existential type schemes:

$$\sigma ::= \tau \mid \texttt{Exist}(\vec{\alpha})\,\tau_1 \to \tau_2 \mid \forall\alpha\cdot\sigma$$

Of course, we consider $\texttt{Exist}(\_)\,\_$ as a binder, when computing free variables and applying substitutions. As for concrete data types in ML, new existential types could be defined to the typing environment by types declarations:

$$\textsf{type}\quad D_i(\tau_i') = K_i \textsf{ of Exist } (\vec{\alpha}_i)\ \tau_i;;$$

where $\vec{\alpha}$ and τ' are linear types (that is, no variable occurs twice) and the union of their free variables contain the free variables of τ. For simplification, we simply assume that the corresponding assertions

$$K_i : \forall\vec{\alpha}_j\cdot\texttt{Exist}(\vec{\alpha}_i)\,\tau_i \to D_i(\tau_i')$$

are in the initial typing environment where α_j's are the free variables of τ and τ' but the α_i's

The syntax of the language is extended with existential introduction and elimination constructs

$$a ::= \ldots \mid K\ a \mid \texttt{let}\ Kx = a_1\ \texttt{in}\ a_2$$

When opening an existential value a_1 as $K\ x$, parts of the type of variable x are abtract in a_2 and cannot be exported outside of the let expression. We assume given a denumerable collection of type symbols Ω that are used to represent abstract parts of types. Typing judgements are modified both to include initial existential type assertions and to introduce Ω type constructors so that their scope can be delimited.

$$A ::= \emptyset \mid A, x:\sigma \mid A, K:\sigma \mid A, \Omega$$

A typing judgement $A \vdash a : \sigma$ is well-formed if all Ω's appearing in type expressions (in σ as well as in A) are introduced in A on the left of the occurrence where they appear (see the appendix 5.4.4). Inference rules are only valid for well-formed typing judgements. We add a constructor introduction rule similar to the VAR rule:

$$\frac{K:\sigma \in A}{A \vdash K : \sigma}$$

Rule INST also applies to K although K is not an expression a. The existential introduction rule is:

$$\frac{A \vdash a : \tau_0 \qquad A \vdash K : \texttt{Exist}(\vec{\alpha})\,\tau_0 \to \tau_1}{A \vdash K\ a : \tau_1}$$

The elimination rule is

$$\frac{\begin{array}{c}A \vdash a_1 : \forall\vec{\alpha}_1\cdot\tau_1 \qquad A \vdash K : \forall\vec{\alpha}_1\cdot\texttt{Exist}(\vec{\alpha}_i)\,\tau_0 \to \tau_1 \\ A, \vec{\Omega}_i[x : \forall\vec{\alpha}_1\cdot\tau_0[\Omega_i(\tau_1)/\alpha_i]] \vdash a_2 : \tau_2\end{array}}{A \vdash \texttt{let}\ K\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

where $\vec{\Omega}$ is a vector of all distinct new constants that do not appear in A. The Ω symbols replace hidden sub-terms of τ_0 that may depend on variables in τ_1. Instead of finding all free variables of τ_1 we simply make all Ω_i's depend on τ_1.

5.2.7 Universal types

Opening an abstract type introduces Ω type symbols with a restricted scope. These type symbols quickly propagate by unification outside of their scope. For instance, the following example is not typable since the argument g is monomorphic and captures the Ω in the type of its argument.

```
type α k = K of Exist (φ) (φ → α) * (φ * φ);;
let fx = K (succ, (0, 1));;

let apply g fp = let (K (f,p)) = fp in f (g p) in apply fst fx;;
| ⟩ Toplevel input:
|⟩let apply g fp = let (K (f,p)) = fp in f (g p) in apply fst fx;;
| ⟩                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
| ⟩ Expression of type β with escaping type ⟨escaped⟩
| ⟩ in type ⟨escaped⟩ * ⟨escaped⟩ of variable p
```

A solution is to pass fst polymorphically and take an instance inside the scope of the de-structuring let expression.

Universal types are dual of existential types. Type scheme are extended to:

$$\sigma ::= \tau \mid \texttt{Exist}(\vec{\alpha})\,\tau_1 \to \tau_2 \mid \texttt{All}(\vec{\alpha})\,\tau_1 \to \tau_2 \mid \forall\alpha\cdot\sigma$$

The initial environment may now also contain assertions of the form:

$$K_i : \forall\vec{\alpha}'\cdot\texttt{All}(\vec{\alpha}_i)\,\tau_i \to D_i(\tau_i')$$

such that $\vec{\alpha}$ and τ' are linear types (no variables may be repeated) and the union of their free variables contains the free variables of τ.

The universal introduction rule is:

$$\frac{A \vdash a : \forall\vec{\alpha}\cdot\tau_0 \qquad A \vdash K : \texttt{All}(\vec{\alpha})\,\tau_0 \to \tau_1}{A \vdash K\ a : \tau_1}$$

If a is not a generalizable expression, $\vec{\alpha}$ will necessarily be empty. This guarantees that mutable data structures are not used polymorphically; the rule applies to universal bindings as well as to let bindings. The elimination rule is

$$\frac{A \vdash a_1 : \tau_1 \qquad A \vdash K : \texttt{All}(\vec{\alpha})\,\tau_0 \to \tau_1 \qquad A, x : \forall\vec{\alpha}\cdot\tau_0 \vdash a_2 : \tau_2}{A \vdash \texttt{let}\ K\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

5.2.8 Mixing the extensions

All extensions informally introduced in previous sections are orthogonal and can be easily mixed together in the language ML-ART. The language is formally described in the appendices.

Type inference had to be omitted by lack of space. Principal type property, and decidability of type inference has been proved for the main extensions of the language taken separately. See [102, 104] for projective types, [30] for recursive record types, [64] for existential types. Other extensions are much simpler, and have obvious type inference algorithms. The combination of recursion with projective types is the one that has to be considered more carefully. Projective types are formalized as terms of an equational algebra. In general, equations do not commute with limits and most equational algebras cannot be extended with recursive types in a trivial way. However the theory of record terms is simple enough to be extended with recursive types [30]. We conjecture

that projective types equality also commutes with limits, and thus can be extended with recursive types.

The language ML-ART is described in the appendix. We give the semantics of the language, and a type-soundness result, but proofs have to be reduced to the main lemmas and a few hints.

5.3 Objects and Inheritance

With the rich type system of ML-ART we can now attempt to define the type of objects. In this section we show how to program objects and inheritance with type abstraction. A first intuitive approach will require unsafe fix points. We then present another solution that avoids any recursion on values.

5.3.1 An intuitive approach to objects and inheritance

The type of an object with exposed representation is of the form $\tau_R * \tau_M$, for some representation type τ_R and a method vector type τ_M where τ_M depends on τ_R. In order to abstract τ_R in τ_M, we need to show this dependence explicitly, at least explicitly enough such that methods can be applied to the representation. We rule that all methods should receive the internal representation as first argument. Thus the type of the method vector is at least a record whose defined fields are abstractions that can be applied to a state of type τ_R. That is, each of the method has type $\tau_R \to \tau_m$ for some type τ_m that may be different on each field. The method vector is always an instance of $\{\alpha.\mathtt{row}\ \tau_R \to \alpha_m\}$ for some representation τ_R. Thus, we define:

```
type ({'attendances.'methods}) object =
    Object of Exist ('R) 'R * {'attendances. ['R] → 'methods};;
```

An object point could be defined as follows. We first define its representation, then its method vector, last we combine the two:

```
let pointR v = {!x = v};;

let pointM =
  let getx R = R.x in
  let setx R x = R.x ← x in
  let move R x = setx R (x + getx R) in
  let print R = print_int (getx R) in
  {getx; setx; move; print};;

let point v = Object (pointR v, pointM);;
```

The move method explicitly uses the getx and setx methods that have been defined simultaneously. If the getx method is later redefined, for instance in:

```
let better_pointM =
  let getx R x = −R.x in let setx R x = R.x ← −x in
  pointM || {getx; setx};;
```

then the move method of "better" points still uses the old getx and setx methods. The right definition of move must take getx and setx dynamically from the object method vector. The well-known solution is to use the so-called "self" methods:

```
let pointM self =
```

```
let getx R = R.x
and setx R x = R.x ← x
and move R x = self.setx R (x + self.getx R)
and print R = print_int (self.getx R) in
{getx; setx; move; print};;
```

The creation of points has to build the recursive method vector as follows:

```
let point v = let rec self = pointM self in Object (pointR v, self);;
```

This kind of recursive definition is not in the core language. The example is safe, but the general case is not. It must checked that the expression pointM self does not access fields of self before they are filled. This is obvious here because pointM does not send any message, but it requires some non trivial analysis in general. In the implementation of ML-ART there is a secret entrance to the unsafe recursion that relies on the programmer to verify the above property by hand. Safer but heavier solutions to this problem are proposed at the end of the paper; another approach to objects that avoids recursion on values is also described further. In the rest of this part unsafe recursion is used without any warning.

The above implementation of better_pointM by re-using methods from pointM produces the expected behavior.

```
let better_pointM self =
  let getx R x = −R.x and setx R x = R.x ← −x in
  pointM self || {getx; setx};;
```

Inheritance is basically sharing of methods, and we have also realized some of it when reusing pointM methods in the definition of better_pointM methods. Extending points with color points requires the extension of the state as well. Hiding the implementation of points, we can define colorR as follows:

```
type color = Blue | Red;;
let colorR superR (c) = superR || {c};;
```

The method vector of color points should add a new method getc to the methods of points. The method print had better be redefined to print the color as well. For instance, it can first print the point as before reusing the print method of points, then print the color. However, it is better to abstract on the method vector of points, called the super class, so that better_points can be extended with color as well.

```
let colorM self super =
  let getc R = R.c
  and print R =
    print_string (match self.getc R with Blue → "blue" | Red → "red"); super.print R in
 super || {getc; print};;
```

Then we create:

```
let color_pointM self = colorM self (pointM self);;
let color_better_pointM self = colorM self (better_pointM self);;
```

Points might themselves have been created from abstract points. In abstract points, only the move method and a default print method are defined. Polar or carthesian points may later implement getx and setx with different representations and still be subclasses of abstract points.

```
let abstract_pointM self =
```

```
  let move R x = self.setx R (x + self.getx R)
  and print R = () in
  {move; print};;
```

Therefore points should also have been defined by abstracting the method vector of their super class. For sake of uniformity, we rewrite all definitions of method vectors by abstracting the super-class method vector. Similarly all representations should abstract the super representation.

```
let pointR superR v = superR || {!x = v};;
let pointM self super =
  super || {getx; setx; move; print};;
```

Real representations and real methods can be recovered anytime by applying the representation function or the method function to empty records.

```
type null = Null;;
let emptyM = ({}: {abs.α → null}) and emptyR = ({}: {abs.null});;
let point v =
  let rec self = pointM self emptyM in  Object (pointR emptyR v, self);;
```

Inheritance is essentially methods sharing, but in a very structured way! Classes are just a way of structuring inheritance.

```
type α class = Class of α;;
```

We group in a same object both components of points. For instance, the empty class is defined as:

```
let nullR superR () = superR;;
let nullM _ super = super;;
let nullC = Class (nullR, nullM);;
```

A natural attempt to write a function new that creates instances of a given class does not work:

```
let new (Class (R,M)) v = let rec self = M self emptyM in Object (R emptyR v, self);;
```

The recursive call is unsafe (though it could be proved correct by hand) but, worse, it cannot easily be compiled correctly. The compiler need to know the size of the recursive value that it is going to create. The difficulty may be avoided by putting the method vector into a fix size top structure, thus using usafe but correctly compiled recursion:

```
type α methods = Methods of α;;
type ({'att.'meths}) object = Object of Exist ('R) 'R * {'att. ['R] → 'meths} methods;;

let new (Class (R,M)) v =
  let rec self = Methods (M self emptyM) in  Object (R emptyR v, self);;

let null = new nullC;;
```

Access to self inside the definition of method vectors must be re-written with one more indirection. The simplest way to send objects messages is to define a send function for each message:

```
let send_getx P = let (Object (R, Methods M)) = P in M.getx R;;
```

Another option is to view messages as field extractors, and define a unique send function:

```
let getx = fun z → z.getx;;
```

Unfortunately the following function fails to type, because of abstraction scope violation:

```
let send extractor object =
  let (Object (R, Methods M)) = object in extractor M R;;
| Toplevel input:
|  let (Object (R, Methods M)) = object in extractor M R;;
|  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
| Expression of type α with escaping type ⟨escaped⟩
| in type {β.[⟨escaped⟩] → γ} of variable M
```

The solution is to make extractors polymorphic on the representation, so that the abstract representation is not exported within the extractor.

```
type ({'att.'meths}, α) extractor = Extractor of All ('R) {'att. ['R] → 'meths} → 'R → α;;

let getx = Extractor (fun z → z.getx);;
let print = Extractor (fun z → z.print);;

let send extractor P =
  let (Object (R, Methods M)) = P in let (Extractor x) = extractor in x M R;;
```

The final version of pointM is:

```
let pointM self super =
  let getx R = R.x
  and setx R x = R.x ← x
  and move R x = send setx (Object (R,self)) (x + send getx (Object (R,self)))
  and print R = print_int (send getx (Object (R,self))) in
  super || {getx; setx; move; print};;

let pointC = Class (pointR, pointM);;

let p1 = new pointC 1 in send move p1 2; send print p1;;
```

Simple inheritance can be done in a systematic way by defining:

```
let inherits (Class (R1, M1)) (Class (R2, M2)) =
  let R superR (v1,v2) = R2 (R1 superR v1) v2 in
  let M self super = M2 self (M1 self super) in
  Class (R, M);;
```

A method can of course return the object that lauches it, or other objects of the same type. For instance, we may define a copy method that returns a new isomorphic object composed of a copy of the state and the same method vector.

```
let copyM self super =
  let copy R = Object (R || {}, self) in
  super || {copy};;
let copyC = Class (nullR, copyM);;
```

Then, providing the copying method to points is quite easy:

```
let copy_pointC() = inherits pointC copyC;;
```

Since classes are polymorphic, we need to delay their assemblage until they are used monomorphically to create objects. The is one example of limitation of polymorphism on values as proposed in [124]. This is quite odd but admissible here since the assemblage may still be done only once before the creation of all objects of the same class.

Our classes are called wrappers in [52]. There are similar to functions that given any class would return a class by adding their own methods. The inherits function simply composes them. while the new function applies them to the empty class before building new objects. Lifting classes to wrappers is basically the same as lifting records to functional records as done in [107]; it provides multiple inheritance for free. A name wrapper can be defined as the color one:

```
let nameR superR (n) = superR || {n};;

let nameM self super =
  let getn R = R.n
  and print R = print_string (send getn (Object (R, self))); print_char ' '; super.print R in
 super || {getn; print};;

let nameC = Class (pointR, pointM);;
```

Named color points can be defined by wrapping points with either color, then name or name then color.

```
let name_color_point() = inherits pointC (inherits colorC nameC);;
let color_name_point() = inherits pointC (inherits nameC colorC);;
```

Both ways are not equivalent; for instance, the last one will print the color before the name, which usually looks very odd. Wrappers have replaced multiple inheritance by single inheritance. It is a more significant example to assume that named-point class and color-point class are defined first, then attempt to implement their combination but ignoring how they were created. The composition

```
let name_color_pointC() = inherits (name_pointC()) (color_pointC());;
```

is not quite correct since the print method is taken from the last class and thus does not print the color. It is usual when doing multiple inheritance that multiply-defined methods need to be redefined. One possibility is to wrap a new print method around the bad definition. Another solution is to do less meta-programming and more programming, that is to define name-color-point methods directly:

```
let color_pointM self super = colorM self (pointM self super)
and name_pointM self super = nameM self (pointM self super);;

let name_color_pointM self super =
  let super1 = color_pointM self super in
  let super2 = name_pointM self super1 in
  let print R = print_string (send getn (Object (R, self))); print_char '='; super1.print R in
  let print_no_name = super1.print in
  super2 || {print; print_no_name};;

let name_color_pointR superR (x,c,n) = superR || {!x = x; c; n};;
let name_color_pointC = Class (name_color_pointR, name_color_pointM);;


let p1 = new name_color_pointC (1, Red, "p") in send print p1;;
p=red1- : unit = ()
```

This is all the flexibility of programmable objects.

5.3.2 Safe fix-points of non-functional values

As shown in the language definition, it is possible to define a fix-point operator on abstractions. Fix point of non-functional values raises difficulties. Their compilation usually requires to know the exact size of the top structure of recursive values; a dummy value of that size is allocated before the evaluation of the recursive definition, whose result is used to patch the dummy value. Thus, at least the top structure of the recursive value must be statically known. The evaluation of non-functional values also assumes that the dummy value is only passed to other functions, stored inside closures, but never accessed before it is patched. This kind of analysis is similar to checking that the evaluation of some expression does not create any reference cell. This problem has been widely addressed recently but has not found any satisfactory solution yet. It can be thought that any good solution for detecting the creation of references can be applied to the detection of unsafe recursions as well.

Simpler solutions could require annotations of the source code to help the static analyzer. There are easy solutions that would automatically guarantee safety of the above examples. All of them are still more or less *ad hoc*, therefore none of them has been included into the language ML-ART.

Another approach is to remark that call-by-name fix points are always safe and not restricted to functions, and that call-by-name can be simulated with call-by-value. That is, recursive values can be replaced by recursive abstractions on values, which can be defined safely. This solution has been proposed in [92]. However, extra abstractions stop evaluation and method vectors are recreated any time a message resends another message to itself, which is too much inefficient. Moreover call-by-value runtime errors (unsafe examples) have been changed into call-by-name "safe" loops. Is this more satisfactory?

5.3.3 Objects without recursive values

There is a very simple way of avoiding recursive objects, which required fix-points of non-recursive values. Going from objects with value abstraction to objects with type abstraction, we moved the abstraction on state from outside the method vector into each method. Similarly, we can move abstraction on self into methods. For instance, the method move can be defined as:

```
let move (R,M) = send setx (Object (R,M)) (x + send getx (Object (R,M)))
```

The function send would correspondingly be changed to

```
let send extractor P = let (Object (R,M)) = P in let (Extractor x) = extractor in x M (R,M);;
```

Since `x M` is a component of M, the expression x M (R,M) contains an application of a part of M to a structure containing itself; thus, it must have a recursive type. Unsurprisingly, the object type must be redefined to:

```
type ({'att.'meths}) object = Object of Exist ('R) rec 'RM in 'R * {'att. ['RM] → 'meths};;
```

The extractor type must also be changed to:

```
type ({'att.'meths}, α) extractor =  Extractor of
  All ('R,'M) {'att. (['R] * 'M) → 'meths} → ('R * {'att. (['R] * 'M) → 'meths}) → α;;
```

Of course, all extractors must be re-evaluated. Sending a message becomes:

```
let send extractor P = let (Object (R,M)) = P in let (Extractor x) = extractor in x M (R,M);;
```

Method vectors are defined by abstracting the super class.

```
let pointM super =
  let getx (R,M) = R.x
  and setx (R,M) x = R.x ← x
  and move P x = send setx (Object P) (x + send getx (Object P))
  and print P = print_int (send getx (Object P)) in
  (super || {getx; setx; move; print});;
let pointC = Class (pointR, pointM);;
```

Since the object constructor is only a type coercion, that is `Object P` and P are the same value, this approach is more efficient than the previous one; it does not sucessively disassemble and re-assemble the state and the method vector.

Creating a new class does not involve recursion any more:

```
let new (Class (R,M)) v = Object (R emptyR v, M emptyM);;
```

For instance,

```
let p1 = new pointC 1 in send print p1;;
1- : unit = ()
```

All examples of the previous section can be re-programmed in the new style, keeping entirely within core ML-ART.

5.3.4 Extensions

We have shown how to program most object constructions. It is lacking the ability to implicitly forget methods and coerce objects to their counterparts in super classes. The same message print can be sent to both points and color points. But points and color points have two incompatible types and can never be stored in the same list. Some languages with sub-typing allow this and would take the common interface of all objects that are mixed in the list as the interface of any object of the list.

In order to be able to forget fields in ML-ART, it would be necessary to give a more general type to the extension primitive:

$$(\_ \parallel \{\ell = \_\}) : \{\ell\colon \alpha_0 \ ; \alpha_1\} \to \alpha_2 \to \{\ell\colon \texttt{static}.\epsilon.\alpha_2 \ ; \alpha_1\}$$

Here, fields of record types should be tripples m, f, τ where m is !, `static`, or a variable and f is `abs`, `pre`, or a variable.

Such typing would be sound. However, to get the benefit of the richer structure, ML should be extended with polymorphic recursion. so that the recursion involved in object construction or message passing can be typed with polymorphic flags. Currentlty, the vector of methods in recursive objects is built as a fix point and can only be assigned a monomorphic type.

In order to allow implicit coercions of objects to their super classes some other kind of polymorphism must be used. Adding sub-typing could be one. Type-checking with non structural sub-types may find a solution along the lines of [4]. Objects with type inference but top-level class definitions have also been studied in [30]. Another interesting investigation, and probably the most promising are type isomorphisms of Di Cosmo [36]. It can be expected that they would allow to turn some `pre` flags into flag variables after the recursive objects have been created.

Conclusion

Programming objects with ML-ART is an interesting experiment, that primarily helps to understand objects in several ways. The unavoidable feature in object-oriented programming is message passing. Polymorphic access is required and it suffices to model very simple objects. The next step involves concealing the internal state of objects, either by value abstraction or type abstraction. Hiding by type abstraction has proved to be both easier and simpler. The concept of inheritance is essentially method sharing in a structured way. Polymorphic record extension is sufficient for simple and multiple inheritance. Slightly less important, classes are just a way of structuring inheritance.

As opposed to the encoding in $F^{\omega}_{\leq}$ that requires a lot of systematic, but still necessary, type information, all our example could be written in a natural ML style. This allows us to assert that no syntactic construct is needed for programming objects in ML-ART. Programmable objects are easier to understand than primitive objects; there is no need to learn a new language, instead object-oriented programming can be discovered progressively.

We have presented two programming styles for objects but other interesting ones can certainly be found. Some of them could be offered in libraries to allow the user to choose the complexity of his objects that is consistent with the level of his problem. A beginner would probably adopt a style from the library while the expert would define his own one.

The language ML-ART is a powerful extension to ML. Record types make declarations of record data structures optional. Recursive types may be quite useful in a few other circumstances, however quantified types, through type declarations, seem to possess the degree of higher order needed in practice; type information, carried by constructors, keeps the language very close to ML and make it as easy to use.

The main limitation of our objects is their inability to coerce objects of their super classes. Improvements of the type system should be made to address this problem, this being contingent, however, on finding a satisfactory solution to the second problem of non-functional recursive values. Both of these problems are interesting and worth further investigation.

Last, but not least, polymorphic records used in ML-ART can be compiled very efficiently. The language itself has been implemented as an extension of Caml-Light.

Acknowledgments

I am indebted to Benjamin Pierce for convincing me that abstract types were the correct approach to objects, and for generating many fruitful discussions. This work evolved as a result of my spring visit to Bell Labs, through seminar discussions on objects. An earliest version of objects was also written during this visit.

Appendix

5.4 Definition of the language ML-ART

Notation For brievity, we will write $\langle\tau\rangle$ for shared templates in this section while we wrote `row` τ in the examples. We also write $\mu\alpha.\tau$ instead of `rec` $\alpha.\tau$.

We formalize an extended language with locations (store adresses) and record values. We assume given a denumerable set of locations. Letter l ranges over locations. Record values are finite maps

from locations to terms. When defined by enumeration, priority is given to the right, that is, in

$$\{\ell_1 = a_1, \ldots \ell_n = a_n, \ell = a\}$$

ℓ may be one of the ℓ_i, but field ℓ is always mapped to a.

The syntax of language expressions, types and typing judgements are defined in figure 5.1.

$a ::= x \mid c \mid \texttt{fun}\ x \to a \mid a_1\ a_2$ $\quad \mid K\ a \mid \texttt{let}\ Kx = a_1\ \texttt{in}\ a_2 \mid \texttt{strip}\ K\ \texttt{of}\ a \mid \{\vec{\ell} = \vec{w}\} \mid l$	Expressions
$c ::= c_a \mid c_b$	Constants
$c_a ::= (\_ \parallel \{!\ell = \_\})$	Dangerous constants
$c_b ::= (\_ \parallel \{\ell = \_\}) \mid (\_.\ell) \mid (\_.\ell \leftarrow \_) \mid \texttt{copy} \mid \ldots$	Safe constants
$b ::= x \mid c_b\ \vec{b} \mid c_a \mid \texttt{fun}\ x \to a \mid \texttt{let}\ Kx = b\ \texttt{in}\ b$ $\quad \mid K\ b \mid \{\vec{\ell} = \vec{w}\}$	Non expansive terms
$v ::= c \mid \texttt{fun}\ x \to a \mid \{\vec{\ell} = \vec{w}\} \mid K\ v$ $w ::= v \mid l$	Values

Figure 5.1: Expressions.

5.4.1 Expressions

The syntax of expressions is given in figure 5.1. Expressions are untyped. Values as a subset of expressions. Non expansive expressions are a subset expressions whose evaluation is guaranteed not to produce any side effect. The type of a non-expansive expression can be generalized.

The following type declarations are not expressions of the language:

$$\mathsf{type}\ k(\tau_0) = K\ \mathsf{of}\ \mathsf{Exist}\ (\vec{\alpha})\ \tau \qquad \text{or} \qquad \mathsf{type}\ k(\tau_0) = K\ \mathsf{of}\ \mathsf{All}\ (\vec{\alpha})\ \tau$$

Instead, they are replaced by type assignment, that for sake of simplicity will be assumed in the initial environment.

$$K : \forall\vec{\alpha}_0 \cdot \texttt{Exist}(\vec{\alpha})\,\tau \to k(\tau_0) \qquad \text{or} \qquad K : \forall\vec{\alpha}_0 \cdot \texttt{All}(\vec{\alpha})\,\tau \to k(\tau_0) \tag{1}$$

We say that value constructor K and type symbol k are paired in type assignment (1). The expression $\texttt{Exist}(\vec{\alpha})\,\tau \to \tau_0$ and $\texttt{All}(\vec{\alpha})\,\tau \to \tau_0$ are well-formed if

- $\vec{\alpha}$ is linear, i.e. no variable occurs twice,
- variables $\vec{\alpha}$ are not in τ_0,
- all variables of τ occur in either $\vec{\alpha}$ or τ_0.

Type scheme $\forall\alpha \cdot \sigma$ is well-formed if σ is. We abbreviate sequences of quantifiers $\forall\alpha_1 \cdot \cdots \cdot \forall\alpha_n \cdot \sigma$ by $\forall\alpha_1, \cdots \alpha_n \cdot \tau$.

The expression $\texttt{strip}\ K\ \texttt{of}\ a$ is used for universal elimination: it strips of the constructor of a value of universal type; the type of the result is an instance of of the universal type associated to K.

Thus, when K is the constructor of a universal data-type, we should now see $\texttt{let}\ K\ x = a_1\ \texttt{in}\ a_2$ (used in section 5.2.6) as syntactic sugar for $\texttt{let}\ x = \texttt{strip}\ K\ \texttt{of}\ a_1\ \texttt{in}\ a_2$. The later form is simpler to formalize. The same simplification cannot be used for existential elimination because the above transformation would break the scope of the type anonymous type symbols introduced by $\texttt{strip}\ K\ \texttt{of}\ a$. Thus, when K is the constriuctor of an existential data-type, the expression $\texttt{let}\ x = Ka_1\ \texttt{in}\ a_2$ both unpacks the existential and generalizes its type within the scope of the let expression. Generalization must happen simultaneously to unpacking, hence we used a "Let" construct. Since the expression $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$ can now be seen as syntactic sugar for $\texttt{let}\ K_0\ x = K_0\ a_1\ \texttt{in}\ a_2$ where $K_0 : \forall\alpha \cdot \texttt{Exist}()\,\alpha \to k_0\alpha$ is in A_0, we removed the original let form from the core language.

5.4.2 Sorts and types

$\tau ::= \alpha \mid \tau \to \tau \mid \mu\alpha.\tau \mid \{\tau\} \mid C(\vec{\tau})$	Types
$\quad \mid \tau.\tau \mid (\ell : \tau; \tau) \mid \texttt{abs} \mid \tau\,\texttt{pre} \mid \texttt{mut} \mid \texttt{static} \mid \langle\tau\rangle$	
$C ::= k \mid D \mid \Omega$	Type constants
$\kappa ::= \texttt{Usual} \mid \texttt{Field} \mid \texttt{Flag} \mid \texttt{Mutability}$	Kind sorts
$\pi ::= \texttt{Type} \mid \texttt{Row}\,(L)$	Power sorts
$L ::= \emptyset \mid \ell.L \qquad \ell \notin L$	

Figure 5.2: Sorts and types.

Symbols	Kinds	Powers
C	$(\texttt{Usual}, \texttt{Usual}) \Rightarrow \texttt{Usual}$	$(\pi, \pi) \Rightarrow \pi$
$(\_ \to \_)$	$(\texttt{Usual}, \texttt{Usual}) \Rightarrow \texttt{Usual}$	$(\pi, \pi) \Rightarrow \pi$
$\{\_\}$	$\texttt{Field} \Rightarrow \texttt{Usual}$	$\texttt{Row}\,(\emptyset) \Rightarrow \texttt{Type}$
$\_.\_$	$(\texttt{Flag}, \texttt{Usual}) \Rightarrow \texttt{Field}$	$(\pi, \pi) \Rightarrow \pi$
$(\ell : \_; \_)$	$(\kappa, \kappa) \Rightarrow \kappa$	$(\texttt{Type}, \texttt{Row}\,(\ell.L)) \Rightarrow \texttt{Row}\,(L)$
abs	Flag	π
_pre	$\texttt{Mutability} \Rightarrow \texttt{Flag}$	$\pi \Rightarrow \pi$
mut	Mutability	π
static	Mutability	π
$\langle\_\rangle$	$\texttt{Usual} \Rightarrow \texttt{Usual}$	$\texttt{Type} \Rightarrow \texttt{Row}\,(L)$

Figure 5.3: Kinds and powers

The syntax of types is described in figure 5.2. Their formation is further restricted twice by kind sorts and power sorts. Kind (respectively power) signatures are non empty sequences of kind (respectively power) sorts, written $\vec{\kappa_i} \Rightarrow \kappa$ or just κ when $\vec{\kappa_i}$ is empty. Each primitive type symbol comes with both a kind signature and a power signature given in figure 5.2. Recursive type expressions ($\mu\alpha.\tau$) requires α and τ to both be of the kind sort Usual and the power sort

`Type`. Type symbols C are universal and existential type symbols k, anonymous type symbols Ω, or regular type symbols D (e.g. type list, int, etc.)

Sort metavariables in signatures mean that all forms ranged over by this meta-variable are possible. Thus, symbols may have several signatures. However, for any term and any sort, there is at most one possible assignment of signatures to symbol occurrences that makes the term well-sorted. There is an algorithm that checks whether such assignment exists and, if so, computes the assignment. Thus it would be possible to work with decorated types, instead, which form a many sorted algebra in the usual meaning.

The most significant sorts are called "kinds": they avoid using flag or mutablility expressions instead of usual types. The other sorts are called "power": expressions of power $\texttt{Row}\,(L)$ are templates in record types and L enumerates all labels that the template must not defined; this is used to avoid redefinition of fields in record types. All types appearing in typing rules and typing environments have the kind `Usual` and the power `Type`.

The above sorts allow such type expressions as $\{\ell_1{:}\,\texttt{abs}.\alpha_1\ ;\ (\ell_2{:}\,\alpha_2.\texttt{abs}).(\alpha_3 \to \alpha_4)\}$ but types that the user may see only uses the weaker kind signature $(\texttt{Field}, \texttt{Field}) \Rightarrow \texttt{Field}$ for $(\ell : \_; \_)$ symbols, which forbids such types as above.

5.4.3 Type equality

We write $[\tau/\alpha]$ the substitution that replaces free occurrences of α by τ. We use letter C to range other all type symbol, and letter f, and g to range other any type symbol other than $(\ell : \_; \_)$, $\langle \_ \rangle$, and $\{\_\}$.

Type equality is the smallest congruence that satisfies the equations of the projective algebra and those for recursive types. The rules for congruence are

$$\frac{}{\tau_1 = \tau_1}\ \text{(Reflexivity)} \qquad \frac{\tau_1 = \tau_2}{\tau_2 = \tau_1}\ \text{(Symmetry)} \qquad \frac{\tau_1 = \tau_2 \qquad \tau_2 = \tau_3}{\tau_1 = \tau_3}\ \text{(Transitivity)}$$

$$\frac{\tau_1 = \tau_1' \qquad \dots \tau_n = \tau_n'}{C(\tau_1, \dots \tau_n) = C(\tau_1', \dots \tau_n')}\ \text{(Congruence)}$$

For sake of simplicity, we will include equivalence rules (Relexivity, Symmetry, and Transitivity) in the notation, and omit them in derivations.

Type equations of the projective algebra are, for any type symbol C other that $(\ell : \_; \_)$, $\langle \_ \rangle$, and $\{\}$, for any labels ℓ_1, ℓ_2, and ℓ,

$$(\ell_1 : \alpha_1; \ell_2 : \alpha_2; \alpha_0) = (\ell_2 : \alpha_2; \ell_1 : \alpha_1; \alpha_0)\quad \text{(Left-commutativity)}$$

$$f\left(\overrightarrow{\ell : \alpha_1; \alpha_2}\right) = (\ell : f(\vec{\alpha}_1); f(\vec{\alpha}_2))\quad \text{(Distributivity)}$$

$$\langle f(\vec{\alpha})\rangle = f\left(\overrightarrow{\langle\alpha\rangle}\right)\quad \text{(Row-distributivity)} \qquad \langle\alpha\rangle = (\ell : \alpha; \langle\alpha\rangle)\quad \text{(Idempotence)}$$

The equational theory of projective types is regular and collapse free, but non linear. It is studied and proved syntactic in [104] in the absence of recursive types. We show below that this result extends to recursive types.

The recursive type expression $(\mu\alpha.\tau)$ is well formed only if both α and τ have the power sort and if τ is neither a type variable, nor another $(\mu\_.\_)$. This guarantees that τ is contractive in α and that $\mu\alpha.\tau$ is well-defined. In practice, when we write types, we may require that α also be of power sort `Type`; this would not be a true restriction since there cannot be cycles along a path

composed only of row symbols. However, we do not impose such a restriction in the formalization. The symbol $\mu\_.\_$ acts as a binder, and in examples, we always assume that bound variables have been renamed properly in order to avoid capture.

Equality for recursive types is taken from [5] (all types are assumed to be well-sorted) and defined by the set of inference rules (μ):

$$\frac{\tau_1 = \tau_2}{\mu\alpha.\tau_1 = \mu\alpha.\tau_2} \quad (\mu\text{-Congruence}) \qquad \mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha] \quad (\text{Fold-Unfold})$$

$$\text{If } \tau \not\equiv \beta \text{ (1) and } \tau \not\equiv \mu\beta.\tau' \text{ (2)}, \qquad \frac{\tau_1 = \tau[\tau_1/\alpha] \wedge \tau_2 = \tau[\tau_2/\alpha]}{\tau_1 = \tau_2} \quad (\text{Contract})$$

The condidtions (1) and (2) of Contract implies that τ is contractive in α. If τ is neither a variable (1) nor another term of the form $\mu\alpha'.\tau'$ (2). These conditions can always be enforced when τ is contractive. In fact, we will furthermore restrict use of rule Contract to cases where $\alpha \in \tau$, since otherwise the rule degenerates into a transitive rule.

Note that rule μ-Congruence is derivable from other rules.

Proof: Prove first that the system without this rule is stable by substitution (Same proof as below, but without case μ-Congruence). Assuming $\tau_1 =_{\mu E} \tau_2$, the equality $\tau_1[\mu\alpha.\tau_2/\alpha] = \tau_2[\mu\alpha.\tau_2/\alpha]$ follows by substitution, and we conclude with the following derivation:

$$\dfrac{\mu\alpha.\tau_1 = \tau_1[\mu\alpha.\tau_1/\alpha] \qquad \dfrac{\mu\alpha.\tau_2 = \tau_2[\mu\alpha.\tau_2/\alpha] \quad \tau_1[\mu\alpha.\tau_2/\alpha] =_{\mu E} \tau_2[\mu\alpha.\tau_2/\alpha]}{\mu\alpha.\tau_2 = \tau_1[\mu\alpha.\tau_2/\alpha]}\ (\text{Transitivity})}{\mu\alpha.\tau_1 =_{\mu E} \mu\alpha.\tau_2}\ (\text{Contract})$$

■

Therefore, in proofs, we always asume that rule μ-Congruence does not appear in derivations.

5.4.4 Typing rules

$\sigma ::= \tau \mid \texttt{Exist}(\vec{\alpha})\,\tau_1 \to \tau_2 \mid \texttt{All}(\vec{\alpha})\,\tau_1 \to \tau_2 \mid \forall\alpha\cdot\sigma$	Type schemes
$A ::= \emptyset \mid A, z:\sigma \mid A, f$	Typing environments
$z ::= x \mid c \mid K \mid l$	
$\Delta ::= A \vdash a:\sigma \mid A \vdash K:\sigma$	Judgements

Figure 5.4: Type schemes and typing judgments

Well-formed typing environments are recursively defined as follows. The empty environment is well-formed. The environment A, Ω is well-formed if A is well-formed and does not introduce Ω. The environment $A, z:\sigma$ is well-formed if A is and if all symbols of σ are predefined or introduced in A. Moreover, if z is a location, then σ must be a simple type τ. Last, type assignment formula $A \vdash \_ : \sigma$ is well-formed if the environment $A, \_ : \sigma$ is. We also asume that all typing contexts are well-sorted, but we omit sorts in the formalization. (In particular, Ω must be used with consistent sorts.)

We assume that the initial environment A_0 assign the types schemes described in figure 5.5 to record constants. We also assume that A_0 contain paired bindings for type constructors of existential or universal types. The conditions that type constructors K_i are paired with a unique type symbol k_i is essential to ensure type soundness. This makes the constructor unique for the corresponding type.

$$
\begin{aligned}
(\_.l) &: \forall\alpha_0,\alpha_1,\alpha_2 \cdot \{\ell\colon \alpha_0\,\mathtt{pre}.\alpha_1\ ;\ \alpha_2\} \to \alpha_1\\
(\_.l \leftarrow \_) &: \forall\alpha_0,\alpha_1,\alpha_2 \cdot \{\ell\colon \mathtt{mut}\,\mathtt{pre}.\alpha_1\ ;\ \alpha_2\} \to \alpha_1 \to \mathtt{unit}\\
(\_ \parallel \{\ell = \_\}) &: \forall\alpha_0,\alpha_1,\alpha_2 \cdot \{\ell\colon \alpha_0\ ;\ \alpha_1\} \to \alpha_2 \to \{\ell\colon \mathtt{static}\,\mathtt{pre}.\alpha_2\ ;\ \alpha_1\}\\
(\_ \parallel \{!\ell = \_\}) &: \forall\alpha_0,\alpha_1,\alpha_2 \cdot \{\ell\colon \alpha_0\ ;\ \alpha_1\} \to \alpha_2 \to \{\ell\colon \mathtt{mut}\,\mathtt{pre}.\alpha_2\ ;\ \alpha_1\}\\
\mathtt{copy} &: \forall\alpha \cdot \{\alpha\} \to \{\alpha\}\\
\\
K_i &: \forall\vec{\alpha}_0 \cdot \mathtt{Exist}(\vec{\alpha})\,\tau \to k_i(\tau_0)\\
K_j &: \forall\vec{\alpha}_0 \cdot \mathtt{All}(\vec{\alpha})\,\tau \to k_j(\tau_0)
\end{aligned}
$$

Figure 5.5: Type schemes of record primitives

$$
\begin{array}{lll}
\sigma ::= \tau \mid \mathtt{Exist}(\vec{\alpha})\,\tau_1 \to \tau_2 \mid \mathtt{All}(\vec{\alpha})\,\tau_1 \to \tau_2 & & \text{Type schemes}\\
\quad \mid \forall\alpha \cdot \sigma & & \\
\\
A ::= \emptyset \mid A, l : \tau \mid A, x : \sigma \mid A, c : \sigma & & \text{Type environments}\\
\quad \mid A, K : \sigma \mid A, \Omega & & \\
\Delta ::= A \vdash a : \sigma \mid A \vdash K : \sigma \mid A \vdash \sigma & & \text{Judgements}
\end{array}
$$

Figure 5.6: Type assignment formulas

We write $FV(A)$ for free variables of A. Type schemes and type assignment formulas are given in figure 5.6. Typing rules are given in figure 5.7. Variable z ranges over identifiers x, c, and K. Rules Exist and All should be seen as existential and universal introduction rules.

In a derivation of a typing judgement, Gen rules can only be used as the last ones or before the left hand sides of Let and Forall rules, since these are the only premises that allow type schemes. We write Gen* for a possibly empty sequence of Gen rules. Moreover, we can always assume that it is used as much as possible on the left hand sides of Let rules. We call such derivations canonical.

In the equational theory that is considered, the arrow type symbol is decomposable, that is:

$$\tau_1 \to \tau_2 = \tau \Longrightarrow \tau \equiv \tau_3 \to \tau_4 \wedge \tau_1 = \tau_3 \wedge \tau_2 = \tau_4$$

As a result, it is always posssible to move rule Equal* around all other typing rules so that it only occurs above left-side premises of App rules and as the last rule of the derivation. For simplicity of presententation, however, we will omit rule Equal* and instead consider types modulo the equational theory. We will only emphasize places where the presence of rule Equal* is essential.

The following properties of typings will be used.

Proposition 5 (Stability by substitution) *If $A \vdash a : \tau$ then $\mu(A) \vdash a : \mu(\tau)$ for any substitution μ such that the formula is well-formed.*

$$\frac{z : \forall\vec{\alpha}_j \cdot \tau \in A}{A \vdash z : \tau[\vec{\tau}_j/\vec{\alpha}_j]} \quad (\textsc{Get}) \qquad \frac{A \vdash a : \tau_1 \qquad \tau_1 = \tau_0}{A \vdash a : \tau_0} \quad (\textsc{Equal}^*)$$

$$\frac{A \vdash b : \sigma \qquad \alpha \notin FV_G(A)}{A \vdash b : \forall\alpha \cdot \sigma} \quad (\textsc{Gen}) \qquad \frac{A, x : \tau_0 \vdash a : \tau_1}{A \vdash \texttt{fun}\ x \to a : \tau_0 \to \tau_1} \quad (\textsc{Fun})$$

$$\frac{A \vdash a_1 : \tau_1 \to \tau_0 \qquad A \vdash a_2 : \tau_1}{A \vdash a_1\ a_2 : \tau_0} \quad (\textsc{App})$$

$$\frac{A \vdash v_1 : \tau_1 \qquad \ldots \qquad A \vdash v_n : \tau_n}{A \vdash \{\ell_1 = v_1, \ldots \ell_n = v_n\} : \{\ell_1\colon \texttt{pre}.\tau_1\ ; \ldots \ell_n\colon \texttt{pre}.\tau_n\ ; \texttt{abs}.\alpha\}} \quad (\textsc{Record})$$

$$\frac{A \vdash a : \tau_0 \qquad A \vdash K : \texttt{Exist}(\vec{\tau})\,\tau_0 \to \tau_1}{A \vdash K\ a : \tau_1} \quad (\textsc{Exist})$$

$$\frac{A \vdash a : \forall\vec{\alpha} \cdot \tau_0 \qquad A \vdash K : \texttt{All}(\vec{\alpha})\,\tau_0 \to \tau_1}{A \vdash K\ a : \tau_1} \quad (\textsc{All})$$

$$\frac{A \vdash a_1 : \tau_1 \qquad A \vdash K : \texttt{All}(\vec{\tau})\,\tau_0 \to \tau_1}{A \vdash \texttt{strip}\ K\ \texttt{of}\ a_1 : \tau_0} \quad (\textsc{Strip})$$

$$\frac{\begin{array}{c} A \vdash K : \forall\vec{\alpha}_1, \vec{\alpha}_j \cdot \texttt{Exist}(\vec{\alpha}_j)\,\tau_0 \to \tau_1 \\ A \vdash a_1 : \forall\vec{\alpha}_1 \cdot \tau_1 \qquad A, \vec{\Omega}_j, x : \forall\vec{\alpha}_1 \cdot \tau_0[\vec{\Omega}_j(\tau_1)/\vec{\alpha}_j] \vdash a_2 : \tau_2 \end{array}}{A \vdash \texttt{let}\ K\ x = a_1\ \texttt{in}\ a_2 : \tau_2} \quad (\textsc{Let})$$

Figure 5.7: Typing rules.

Proposition 6 (Extension of environment) *If the type-assignement A and B are identical everywhere except maybe on variables that are not free in a, then $A \vdash a : \sigma$ is derivable if and only if $B \vdash a : \sigma$ is.*

These properties are proved in [102] for core ML when types are taken modulo a regular equational theory. A regular theory is one such that two equal terms always have the same free variables. All equations for the projective algebra and for recursive types are regular. The proofs of [102] easily extend to the language ML-ART.

In [102] we also show that the language has principal typings if the equational theory has principal unifiers, and that type inference then reduces to unification. Although not stated explicitly for recursive types, the results and proof of [102] extend to the case of recursive types. The proofs also extend to the new constructs of ML-ART. Thus, the algorithm for type inference without recursive types is sound and complete in the presence of recursive types.

5.5 Syntactiness of recursive projective types and type inference

The aim of this section is to show the syntacticness of the equational theory presented in the previous section. We show that the axioms of the theory E form a syntactic presentation for μ-equality. This means that the obvious mutation rules derived from the axioms (given in figure 5.9)

preserve sets of unifiers (i.e. they are sound and complete) for μE-equality. This was proved in E in [103], but only for E-equality (no recursive types). The proof used results from [104]; it does not easily extend to μE-equality (recursive types). Here, we provide a direct proof of syntacticness, reusing the idea of canonical forms introduced in [100].

$$\frac{\alpha \doteq e \wedge \alpha \doteq e'}{\alpha \doteq e \doteq e'} \quad \textsc{Fuse}$$

$$\frac{C(\tau_i)^{i\in 1,p} \doteq C(\tau'_i)^{i\in 1,p} \doteq e}{\exists(\alpha)^{i\in 1,p} \cdot \bigwedge \begin{cases} \wedge f(\alpha_i)^{i\in 1,p} \doteq e \\ \alpha_i \doteq \tau_i \doteq \tau'_i \qquad i \in [1,p] \end{cases}} \quad \textsc{Decompose}$$

$$\frac{f(\tau_i)^{i\in 1,p} \doteq f'(\tau'_j)^{j\in 1,q} \doteq e}{\bot} \quad \textsc{Collision}(f,f') \qquad \text{if } f \neq f'$$

Figure 5.8: Rules for unification in the empty theory.

In this part we write τ_i for a tuple of types. The indice i range over a set I, usually left implicit from the context. By default, we choose those indices distinct from integers, so that, when i is implicit, τ_1 and τ_i are independant types, i.e. τ_1 is not τ_i for some i. We write $=_{\mu E}$ for μE-equality in the equational theory (or simply $=$ when there is no ambiguity) and $\equiv$ for textual equality.

As usual, substitutions are sort-respecting mappings from variable to types that are the identity almost everywhere. Substitutions are extended to mappings from types to types by congruence. Therefore $(\mu\alpha.\tau)[\tau_0/\alpha_0]$ is by definition equal to $\mu\alpha.(\tau[\tau_0/\alpha_0])$ (by naming convention α and α_0 are dinstinct).

Lemma 7 *The relation $=_{\mu E}$ is stable by substitution.*

Proof: We show the lemma for a substitution $[\tau_0/\alpha_0]$. It follows for more genral substitutions by composition into smaller substitutions. By induction on the size of the derivation. We assume given a derivation of $\tau_1 =_{\mu E} \tau_2$ and that any substitution of provably equal term with a strictly smaller derivation are equal. We show $\tau_1[\tau_0/\alpha_0] =_{\mu E} \tau_2[\tau_0/\alpha_0]$.

Case Congruence: trivial.

Case Axioms: immediate, since the axioms are closed by substitutions.

Case Fold-Unfold**:** τ_1 is $\mu\alpha.\tau$ and τ_2 is $\tau[\tau_1/\alpha]$ and are equal. We asume that α does not appear in τ_0. Terms $\tau_1[\tau_0/\alpha_0]$ and $\tau_2[\tau_0/\alpha_0]$ are respectively $\mu\alpha.(\tau[\tau_0/\alpha_0])$ and $(\tau[\tau_0/\alpha_0])[\tau[\tau_0/\alpha_0]/\alpha]$, and therefore are equal by rule Fold-Unfold axiom.

Case Contract**:** τ_1 and τ_2 are $\tau[\tau_1/\alpha]$ and $\tau[\tau_2/\alpha]$. Then, assuming that α_0 and α are distinct and that α is not free in τ_0, $\tau_1[\tau_0/\alpha_0]$ is $\tau[\tau_0/\alpha_0][\tau_1[\tau_0/\alpha_0]/\alpha]$ and similarly $\tau_2[\tau_0/\alpha_0]$ is $\tau[\tau_0/\alpha_0][\tau_2[\tau_0/\alpha_0]/\alpha]$, therefore, by rule Contract, $\tau_1[\tau_0/\alpha_0]$ and $\tau_2[\tau_0/\alpha_0]$ are equal. If α and α_0 are equal or α appears in τ_0, we may replace α by α' and τ by $\tau[\alpha'/\alpha]$ in both premisses. ■

Two substitutions are μE-equal if they have the same domain and the same variables are mapped to μE-equal terms. This property of the definition extends to substitutions of terms:

$$\frac{a^L : \tau; \tau' \doteq b^L : \alpha; \alpha' \doteq e}{\exists \alpha'' \cdot \bigwedge \begin{cases} b^L : \alpha; \alpha' \doteq e \\ \tau \doteq b^a.L : \alpha; \alpha'' \\ \alpha' \doteq a^b.L : \tau; \alpha'' \end{cases}} \quad \textsc{Mutate}(a, b)$$

$$\frac{a^L : \tau; \tau' \doteq \langle \alpha \rangle^L \doteq e}{\langle \alpha \rangle^L \doteq e \wedge \tau \doteq \alpha \wedge \tau' \doteq \langle \alpha \rangle^{a.L}} \quad \textsc{Mutate}(a, \texttt{Row})$$

$$\frac{f^{\texttt{Row}(L)}(\tau_i)^{i \in 1,p} \doteq a^L : \alpha; \alpha' \doteq e}{\exists \alpha_i, \alpha_i'^{i \in 1,p} \cdot \bigwedge \begin{cases} a^L : \alpha; \alpha' \doteq e \\ \alpha_0 \doteq f^{\texttt{Type}}(\alpha_i)^{i \in 1,p} \\ \alpha' \doteq f^{\texttt{Row}(a.L)}(\alpha_i')^{i \in 1,p} \\ \tau_i \doteq a^L : \alpha_i; \alpha_i' \qquad i \in [1,p] \end{cases}} \quad \textsc{Mutate}(f, a)$$

$$\frac{f^{\texttt{Row}(L)}(\tau_i)^{i \in 1,p} \doteq \langle \alpha \rangle^L \doteq e}{\exists (\alpha)^{i \in 1,p} \cdot \bigwedge \begin{cases} \langle \alpha \rangle^L \doteq e \\ \alpha \doteq f^{\texttt{Type}}(\alpha_i)^{i \in 1,p} \\ \tau_i \doteq \langle \alpha_i \rangle^L \qquad i \in [1,p] \end{cases}} \quad \textsc{Mutate}(f, \texttt{Row})$$

Figure 5.9: Mutation rules in the generic algebra of record terms

Lemma 8 *Two μE-equal substitutions map the same term to two equal terms*

Proof: By induction on the size of the term τ, using congruence. ■

We define the top symbol Topτ of a type τ to be C if τ is $C(\tau_1, \ldots \tau_n)$ or Top(τ_1) if τ is $\mu\alpha.\tau_1$. It is undefined for variables. We define the projection $\tau_{/i}$ as τ_i if τ is $C(\tau_1, \ldots \tau_i, \ldots \tau_n)$ and $(\tau_{1/i})[\tau_1/\alpha]$ if τ is $\mu\alpha.\tau_1$. It is undefined if τ is a variable or i is higher than the arity of Top(τ).

Lemma 9 *Two μE-equal non variables terms of the power* `Type` *have the same top symbols and the same projections.*

Proof: By induction on the size of the equality derivation and case analysis on the last rule applied. All cases are easy, in particular since no axiom has power `Type`. ■

On row terms, we define the *projection* $\tau_{/a}$ on label a, the *restriction* $\tau_{\backslash a}$ on label a, the *main* symbol Main(τ), and the extended projection $\tau_{//i}$, inductively on the size of the term and by cases

on its top structure:

τ	$\tau_{/a}$	$\tau_{\backslash a}$	$\mathrm{Main}(\tau)$	$\tau_{//i}$
$(a:\tau_1;\tau_2)$	τ_1	τ_2	$\mathrm{Top}(\tau_1)(1)$	$(a:\tau_{1/i};\tau_{2//i})$ (1)
$(b:\tau_1;\tau_2)$	$\tau_{2/a}$	$(b:\tau_1;\tau_{2\backslash a})$	$\mathrm{Top}(\tau_1)(1)$	$(b:\tau_{1/i};\tau_{2//i})$ (1)
$f(\tau_i)$	$f(\tau_{i/a})$	$f(\tau_{i\backslash a})$	f	τ_i
$\langle\tau_1\rangle$	τ_1	$\langle\tau_1\rangle$	$\mathrm{Top}(\tau_1)$	$\langle(\tau_{1/i})\rangle$

(1) We restrict the definition of $\mathrm{Main}(\tau)$ and $\tau_{//a}$ in the two first lines to cases where $\mathrm{Main}(\tau)$ and $\mathrm{Main}(\tau_{/a})$ are equal. The projection on a and restriction on a are always defined (for well-typed row terms).

The definition is well-founded, since recursive definitions are always called on immediate subterms of the original term, and there is no infinite path composed of only symbols of the power sort `Row`. Therefore, a result answer or the undefined answer will be reached in a finite number of iterations.

Lemma 10 *The above operations are compatible with μE-equality, i.e. given μE-equal row terms their projections (respectively restrictions and extended projection) are either both undefined or μE-equal, and their main symbols are either both undefined or identical.*

Proof: By induction on the size of the proof of equality, then by cases on the last rule applied. We write τ' and τ'' for the two μE-equal terms. We write $\#(\tau)$ any of the operations $\tau_{/a}$, $\tau_{\backslash a}$, or $\tau_{//i}$, but used consistently in the context.

We first notice that whenever $\#\tau$ is defined, then $\#(\theta(\tau))$ is $\theta(\#\tau)$ (substituvity of the definition).

Case Congruence: By congruence, the top symbols of τ' and τ'' are the same, and the direct subterms of τ' and τ'' are μE-equal (which a smaller derivation). By induction, their #-projections are also μE-equal. By congruence, we can rebuild two versions composed of the corresponding subterms that are μE-equal. This shows that $\#\tau'$ and $\#\tau''$ for any #-projection.

Case Fold-Unfold: This is immediate since the definition of each operators is defined by unfolding: $\#(\mu\alpha.\tau)$ is defined as $\#(\tau[\mu\alpha.\tau/\alpha])$. Similarly for Main.

Case Axioms of E: All cases are easy, since the definitions have been written exactly to commute with E-equality.

Case Contract: Assume that $\tau^\epsilon =_{\mu E} \tau[\tau^\epsilon/\alpha]$ when $^\epsilon$ is $'$ or $''$. Assume that α does appear in τ. Since α has power sort `Type`, the operation # is defined on τ^ϵ whenever it is defined on τ. Then it is also defined on τ''. The terms τ^ϵ and $\tau[\tau^\epsilon/\alpha]$ are μE-equal. By induction hypothesis, they remain μE-equal by any #-projection. Therefore $\#\tau^\epsilon$ and $(\#\tau)[\tau^\epsilon/\alpha]$ are μE-equal. Since by hypothesis, τ' and τ'' are μE-equal, the two substitutions $[\tau'/\alpha]$ and $[\tau''/\alpha]$ are also μE-equal; hence, so are $\#\tau'$ and $\#\tau''$.

Othercases: all operations are undefined.

■

Corollary 11 *It immediately follows that*

(E1) If $(a:\tau_1;\tau_2) =_{\mu E} (a:\tau_3;\tau_4)$ *then* $\tau_1 =_{\mu E} \tau_3$ *and* $\tau_2 =_{\mu E} \tau_4$

(E2) If $f(\tau_i) =_{\mu E} f(\tau_i')$ *then* $\tau_i =_{\mu E} \tau_i'$.

The properties $(E1)$, $(E2)$ ensures that standard decomposition rules remain complete for μE-equality.

Lemma 12 *A row term* τ *for which both the projection on label* a *and the restriction on label* a *are defined is* μE*-equal to* $(a:\tau_{/a};\tau_{\backslash a})$.

Proof: By induction on the size of τ and cases on the top symbol of τ

Case $(a:\tau_1;\tau_2)$**:** By definition τ_1 and τ_2 are $\tau_{/a}$ and $\tau_{\backslash a}$.

Case $(b:\tau_1;\tau_2)$**:** By definition $(a:\tau_{/a};\tau_{\backslash a})$ is equal to $(a:\tau_{2/a};(b:\tau_1;\tau_{2\backslash a}))$, which by left commutativity is μE-equal to $(b:\tau_1;(a:\tau_{2/a};\tau_{2\backslash a}))$. By induction hypothesis the right subterm is equal τ_2 and by congruence the whole term is μE-equal to τ.

Case $\langle\tau_1\rangle$**:** By definition, $(a:\tau_{/a};\tau_{\backslash a})$ is equal to $(a:\tau_1;\langle\tau_1\rangle)$, which by the idempotence axiom τ is μE-equal to $\langle\tau_1\rangle$.

Case $f(\tau_i)$**:** By definition $(a:\tau_{/a};\tau_{\backslash a})$ is equal to $(a:f(\tau_{i/a});f(\tau_{i\backslash a})$, which by distributivity is μE-equal to $f(a:\tau_{i/a};\tau_{i\backslash a})$. By induction hypothesis, each subterm $(a:\tau_{i/a};\tau_{i\backslash a})$ is μE-equal to τ_i, thus by f-congruence, it is μE-equal to τ. ■

Corollary 13

(E3) If $(a:\tau_1;\tau_2) =_{\mu E} (b:\tau_3;\tau_4)$ *then there exists a term* τ_5 *such that* $\tau_2 =_{\mu E} (b:\tau_1;\tau_5)$ *and* $\tau_4 = (a:\tau_2;\tau_5)$.

(E4) If $(a:\tau_1;\tau_2)$ *is equal to* $\langle\tau\rangle$*, then* $\tau_1 =_{\mu E} \tau$ *and* $\tau_2 =_{\mu E} \langle\tau\rangle$.

This ensures the completeness of the first two mutation rules.

Lemma 14 *A row term* τ *with extended top symbol* f *is* μE*-equal to* $f(\tau_{//i}{}^{i\in I})$.

Proof: By induction on the size of τ and case analysis on the top symbol of τ.

Case $(a:\tau_1;\tau_2)$**:** Then f is the top symbol of τ_1 and the extended top symbol of τ_2. Therefore, by induction hypothesis τ_2 is equal to $f(\tau_{2//i}{}^{i\in I})$. Since by definition, τ_1 is μE-equal to $f(\tau_{1/i}{}^{i\in I})$, the term τ is μE-equal to $(a:f(\tau_{1/i});f(\tau_{2//i}))$. By distributivity it is μE-equal to $f((a:\tau_{1/i};\tau_{2//i})^{i\in I})$ which is by definition $f(\tau_{//i})$.

Case $\langle\tau_1\rangle$**:** Then $\mathrm{Top}(\tau_1)$ is f and $f((\langle\tau_1\rangle)_{//i}{}^{i\in I})$ is $f(\langle(\tau_{1/i})\rangle^{i\in I})$. This is E-equal to $\langle(f(\tau_{1/i}{}^{i\in I}))\rangle$, i.e. $\langle\tau\rangle_1$.

Case $g(\tau_i)$**:** Then g must be f and there is nothing to prove.

■

Corollary 15

(E5) If $(a:\tau_1;\tau_2) =_{\mu E} f(\tau_i)$*, then there exists* τ_i'*,* τ_i''*, and* τ_3 *such that* $\tau_i =_{\mu E} (a:\tau_i';\tau_i'')$ *and* $\tau_1 =_{\mu E} f(\tau_i')$ *and* $\tau_2 = f(\tau_i'')$.

(E6) If $f(\tau_i) =_{\mu E} \langle\tau\rangle$*, then* $\mathrm{Top}(t)$ *is* f *and* $\tau_i =_{\mu E} \langle\tau_{/i}\rangle$.

This ensures the completeness of the last two mutation rules.

Proof:

Case E5: The term $(a : \tau_1; \tau_2)$ has main symbol f, thus it is μE-equal to $f(a : \tau_{1/i}; \tau_{2//i})$ (1). By distributivity, it is μE-equal to $(a : f(\tau_{1/i}); f(\tau_{2//i}))$ (2).

From (1), we get by property (E2) that $\tau_i =_{\mu E} (a : \tau_{1/i}; \tau_{2//i})$. Form (2), we get by property (E1) that $\tau_1 =_{\mu E} f(\tau_{1/i})$ and $\tau_2 =_{\mu E} f(\tau_{2//i})$.

Case E6: The term $\langle\tau\rangle$ has main extended top symbol f, thus it is μ-equal to $f(\langle\tau_{/i}\rangle)$. By property (E2), we get that τ_i is equal to $\langle\tau_{//i}\rangle$.

■

In summary, we have shown that decomposition and mutation rules are complete. They are sound by constructions. Thus, each of them preserves μE-equality. Hence the axioms E form a syntactic presentation of the theory of recursive projective types. Rules of figure 5.9 together with the rules for unification in the empty theory 5.8 provide an algotithm for μE-unification when the input problem does not have recursive types. To allow recusive types as input, we simply add the rule

$$\frac{\mu\alpha.\tau \doteq e}{\exists\alpha.\alpha \doteq \tau \doteq e} \ (\mu\text{-DECOMPOSE})$$

(since μ and $\exists$ are bindindes, this implicitly assume that α is not free in e.) The rule μ-Decompose is sound and complete. For soundness, assume that S is a solution of the conclusion. Let S' be the restriction of S outside of α. We can derive:

$$\frac{S(\tau) =_{\mu E} S'(\tau)[S(\alpha)/\alpha] \qquad \mu\alpha.S'(\tau) =_{\mu E} S'(\tau)[\mu\alpha.S'(\tau)/\alpha]}{S(\tau) =_{\mu E} \mu\alpha.S'(\tau)} \ (\text{CONTRACT})$$

Since $\mu\alpha.S'(\tau)$ is $S(\mu\alpha.\tau)$, by transitivity with $S(\tau) =_{\mu E} S(e)$, S is a solution of the premise. For completeness, asume that S is a solution of the premisse. We can always assume that S does not affect α. Since $\mu\alpha.\tau$ is μE-equal to $\tau[\mu\alpha.\tau/\alpha]$, the stability of μE-equality by substitution implies that $S(\mu\alpha.\tau)$ is μE-equal to $S(\tau[\mu\alpha.\tau/\alpha])$ (1). Let S' be $S + [S(\mu\alpha.\tau/\alpha]$. The equality (1) can be rewritten into $S(\alpha) =_{\mu E} S(\tau)$, hence the conclusion.

Since rule μ-Decompose strictly decreases the number of μ symbols, it can always be applied first to transform a system of multi-equations with recursive types into an μE-equivalent system of multi-equations without recursive types. The application of other rules in any order terminates, since they are the same set of rules that for E-unification (see [104, 103]) but the occur check has been removed. In fact, since no rules actually introduces recursive types, rule μ-Decompose does not need to be applied first, and can be freely combined with other rules. Since no rule introduces a disjunction, we deduce as a corollary that unification for recursive projective types has principal unifiers. (A principal unifier can straighforwardly be read from a system of multi-equations in canonical form.)

5.6 Semantics

We give a call-by-value store reduction semantics to ML-ART, using the formalism of [47]. Evaluation of programs is defined on pairs a/s of an expression a and a store s. The evaluation is defined by redex rules and an congruence rule allowing reduction in any evaluation context. Stores and evaluations contexts are defined in the figure 5.10. The small-step reduction relation $\longrightarrow$ is defined in in figure 5.11.

We say that store s agrees with typing A, and we write $\Vdash s : A$ if both s and A have the same location domains, and for any location l of their domain $A \vdash s(l) : A(l)$. We call a store extension of A any extension of A with location typing assertions $(l : \tau)$. We write $a_1/s_1 \subset a_2/s_2$ if

$$s ::= [l_1 \mapsto v_1, \ldots l_2 \mapsto v_2] \qquad \text{Stores}$$

$$E ::= [] \mid E\ a \mid v\ E \mid \texttt{let}\ K\ x = E\ \texttt{in}\ a \mid K\ E \mid \texttt{strip}\ K\ \texttt{of}\ E \qquad \text{Evaluation contexts}$$

Figure 5.10: Values, stores and evaluation contexts

$$(\texttt{fun}\ x \to a)\ v/s \longrightarrow_\epsilon a[v/x]/s \qquad \textsc{Fun}$$

$$\texttt{let}\ K\ x = K\ v\ \texttt{in}\ a/s \longrightarrow_\epsilon a[b/x]/s \qquad \textsc{Let}$$

$$\texttt{strip}\ K\ \texttt{of}\ Kv/s \longrightarrow_\epsilon v/s \qquad \textsc{Strip}$$

$$\{\vec{\ell_i} = \vec{w_i}\} \parallel \{!\ell = v\}/s \longrightarrow_\epsilon \texttt{copy}\ \{\vec{\ell_i} = \vec{w_i}, \ell = l\}/s, l \mapsto v \quad \text{if } l \notin s, l \notin \vec{\ell_i} \qquad \textsc{Extend-M}$$

$$\{\vec{\ell_i} = \vec{w_i}\} \parallel \{\ell = v\}/s \longrightarrow_\epsilon \texttt{copy}\ \{\vec{\ell_i} = \vec{w_i}, \ell = v\}/s \quad \text{if } \ell \notin \vec{\ell_i} \qquad \textsc{Extend-S}$$

$$\{\ell = w, \vec{\ell_i} = \vec{w_i}\} \parallel \{!\ell = v\}/s \longrightarrow_\epsilon \texttt{copy}\ \{\vec{\ell_i} = \vec{w_i}, \ell = l\}/s, l \mapsto v \quad \text{if } l \notin s \qquad \textsc{Override-M}$$

$$\{\ell = w, \vec{\ell_i} = \vec{w_i}\} \parallel \{\ell = v\}/s \longrightarrow_\epsilon \texttt{copy}\ \{\vec{\ell_i} = \vec{w_i}, \ell = v\}/s \qquad \textsc{Override-S}$$

$$\{\ldots, \ell = v\}.\ell/s \longrightarrow_\epsilon v/s \quad \text{if } l \in s \qquad \textsc{Dot-S}$$

$$\{\ldots, \ell = l\}.\ell/s \longrightarrow_\epsilon s(l)/s \quad \text{if } l \in s \qquad \textsc{Dot-M}$$

$$\{\ldots, \ell = l\}.\ell \leftarrow v/s \longrightarrow_\epsilon v/s, l \mapsto v \quad \text{if } l \in s \qquad \textsc{Mute}$$

$$\texttt{copy}\ \{\vec{\ell_i} = \vec{l_i}, \vec{\ell_j} = v_j\}/s \longrightarrow_\epsilon \{\vec{\ell_i} = \vec{l_i'}, \vec{\ell_j} = \vec{v_j}\}/s, \vec{l_i'} = s(\vec{l_i}) \quad \text{if } \vec{l_i'} \notin s \qquad \textsc{Copy}$$

$$E[a_1]/s_1 \longrightarrow E[a_2]/s_2 \quad \text{if } a_1/s_1 \longrightarrow_\epsilon a_2/s_2 \qquad \textsc{Context}$$

Figure 5.11: Reduction-rules

- for any environment A_1, any type τ such that $A_1 \vdash a_1 : \tau$ and $\Vdash s_1 : A_1$, there exists a store extension A_2 of A_1 such that $A_2 \vdash a_2 : \tau$ and $\Vdash s_2 : A_2$,
- a_2 is non expansive whenever a_1 is and then A_2 may be chosen equal to A_1.

The soundness of the semantics is formalized by the two following theorems:

Theorem 11 (Subject Reduction) *If $a_0/s_0 \longrightarrow a/s$ then $a_0/s_0 \subset a/s$.*

Theorem 12 (Normal forms) *Let A be a store extension of the initial environment A_0. If $A \vdash a : \tau$ and $\vdash s : A$ and a/s is in $\longrightarrow$-normal form, then a is a value.*

The second theorem asserts that well typed terms that cannot be reduced are values, thus the evaluation is never stuck. It is proved by structural induction on the normal term, and by case analysis on the top structure of the type. Their is no difficulty but, at the difference of ML, types have to be taken modulo the axioms.

Subject reduction is a straightforward combination of redex contration and context replacement lemmas given below.

Lemma 16 (Context replacement) *For any one-hole context E, if $a_1/s_1 \subset a_2/s_2$ then $E[a_1]/s_1 \subset E[a_2]/s_2$.*

By construction, the relation $\subset$ is reflexive, transitive; context replacement says that it is also increasing. The lemma is proved independently for each one-nod context, then the general case follows by induction on the size of the context.

Proof: Let E be a one-nod context. Let A_1 be a type environment and τ a type such that $A \vdash E[a_1] : \tau$ (1) and $\Vdash \sigma_1 : A_1$. It follows from the definition of generalizable terms, that if $E[a_1]$ is generalizable term then so are, successively, a_1, a_2, and $E[a_2]$. We show that there exists a store extension A_2 of A_1 such that $A_2 \vdash a_2 : \tau$ (2) and $\Vdash \sigma_2 : A_2$.

Case E is `let` $K\ x = []$ `in` a: A canonical derivation of (1) ends as:

$$\dfrac{\dfrac{(3)\ \ A_1 \vdash a_1 : \tau_1}{A_1 \vdash a_1 : \forall \vec{\alpha}_1 \cdot \tau_1}\ (\textsc{Gen}^*) \qquad \begin{array}{l} A_1 \vdash K : \forall \vec{\alpha}_1, \vec{\alpha}_j \cdot \mathtt{Exist}(\vec{\alpha}_j)\, \tau_0 \to \tau_1 \\ A_1, \vec{\Omega}_j, x : \forall \vec{\alpha}_1 \cdot \tau_0[\vec{\Omega}_j(\tau_1)/\vec{\alpha}_j] \vdash a : \tau \end{array}}{A_1 \vdash \mathtt{let}\ K\ x = [a_1]\ \mathtt{in}\ a : \tau}\ (\textsc{Let})$$

By the induction hypothesis applied to (3), we know that there exists a store extension A_2 of A_1 such that $A_2 \vdash a_2 : \tau_1$. If a_1 is non expansive then so is a_2 and A_2 may be chosen equal to A_1. Otherwise, the sequence of Gen rules is empty and the sequence of α_1 is also empty. In both cases, we can prove $A_2 \vdash a_2 : \forall \vec{\alpha}_1 \cdot \tau_0$. In the right premises, we may extend the context replacing A_1 by A_2. We conclude by applying rule Let.

Case E is K $[]$: A canonical derivation of (1) may end as:

$$\dfrac{A_1 \vdash a_1 : \tau_1 \qquad A_1 \vdash K : \mathtt{Exist}(\vec{\tau}_2)\, \tau_1 \to \tau}{A_1 \vdash K\ a_1 : \tau}\ (\textsc{Exist})$$

By induction hypothesis applied to the premisse there exists and extension A_2 of A_1 such that $A_2 \vdash a_2 : \tau_1$. We can extend the context of the right premisse and conclude with an Exist rule.

Otherwise, a canonical derivation of (1) ends as:

$$\dfrac{\dfrac{(7)\ \ A_1 \vdash a_1 : \tau_1}{A_1 \vdash a_1 : \forall \vec{\alpha}_1 \cdot \tau_1}\ (\textsc{Gen}^*) \qquad A_1 \vdash K : \mathtt{All}(\vec{\alpha}_1)\, \tau_1 \to \tau}{A \vdash K\ a_1 : \tau}\ (\textsc{All})$$

We reason as for the Let case and to show that there exists an extension A_2 of A_1 such that $A_2 \vdash a_2 : \forall \vec{\alpha}_1 \cdot \tau_1$. We can extend the context of the right premisse and conclude with a All rule.

In both cases, A_2 may be taken equal to A_1 if a is generalizable.

Case E is `strip` K `of` a_1: A canonical derivation of (1) of the form:

$$\dfrac{A_1 \vdash a_1 : \tau_1 \qquad A_1 \vdash K : \mathtt{All}(\vec{\alpha}_2)\, \tau_2 \to \tau_1}{A_1 \vdash \mathtt{strip}\ K\ \mathtt{of}\ a_1 : \tau}\ (\textsc{Strip})$$

where τ is of the form $\tau_2[\tau_i/\alpha_i]$. The induction hypothesis applied the left premisse shows that there exists an extension A_2 of A_1 such that $A_2 \vdash a_2 : \tau_1$. We can extend the context of the right premisse and conclude with a Strip rule. The context A_2 may be taken equal to A_1 if a is generalizable.

Case E is $E\ a$ or $v\ E$: They are similar to the previous case.

Othercases: The expression $E[a]$ is typed as the application of a primitive; these are sub-cases of $v\ E$.

■

Lemma 17 (Redex contraction) *If $a_1/s_1 \longrightarrow_\epsilon a_2/s_2$ then $a_1/s_1 \subset a_2/s_2$.*

The proof can be done independently for each redex. All cases are easy once we have proven the right lemmas.

Lemma 18 (Term replacement) *If the formulas $A \vdash v : \forall\alpha_0 \cdot \tau_0$ (1) and $A, x : \forall\alpha_0 \cdot \tau_0 \vdash a : \tau$ (2) are provable and if bound variables of a are not free in b, then $A \vdash a[b/x] : \tau$ (3) is provable.*

Proof: The proof is by induction on the structure of a. We write A_0 for $A, x : \forall\vec{\alpha}_0 \cdot \tau_0$.

Case a is x: The terms $a[v/x]$ and b are equal. From (2), we know that τ is τ_0 where all variables $\vec{\alpha}_0$ have been substituted. Stability by substitution applied to (1) proves (3).

Case a is y: The term $a[v/x]$ and a are equal.

Case a is $\texttt{let}\ Ky = a_1\ \texttt{in}\ a_2$: A canonical derivation of (2) ends as:

$$\dfrac{\dfrac{(3)\ A_0 \vdash a_1 : \tau_1}{A_0 \vdash a_1 : \forall\alpha_1 \cdot \tau_1}\ (\textsc{Gen}^*) \qquad \begin{array}{c} A_0 \vdash K : \forall\vec{\alpha}_1, \vec{\alpha}_j \cdot \texttt{Exist}(\vec{\alpha}_j)\,\tau_2 \to \tau_1 \\ (4)\ A_0, \vec{\Omega}_j, y : \forall\vec{\alpha}_1 \cdot \tau_2[\vec{\Omega}_j(\tau_1)/\vec{\alpha}_j] \vdash a_2 : \tau \end{array}}{A_0 \vdash \texttt{let}\ Ky = a_1\ \texttt{in}\ a_2 : \tau}\ (\textsc{Let})$$

By induction hypothesis applied to (3), we get $A \vdash a_1[v/x] : \tau_1$, which can be followed by the same Gen rules.

Since y is not free in v, we have $A, \vec{\Omega}_j, y : \forall\alpha_1 \cdot \tau_1 \vdash b : \forall\alpha_0 \cdot \tau_0$ (5) by extension of environment lemma. The induction hypothesis applied to (5) and (4) (in which the order of assignment may be exchanged) proves

$$A, \vec{\Omega}_j, y : \forall\vec{\alpha}_1 \cdot \tau_0[\vec{\Omega}_j(\tau_1)/\vec{\alpha}_j] \vdash a_2[v/x] : \tau$$

We conclude by a Let rule.

Case a is $K\ a_1$ and K: A canonical derivation of (2) may end as:

$$(6)\ \dfrac{A_0 \vdash a_1 : \tau_1 \qquad A_0 \vdash K : \texttt{Exist}(\vec{\tau}_2)\,\tau_1 \to \tau}{A_0 \vdash K\ a_1 : \tau}\ (\textsc{Exist})$$

Applying the induction hypothesis to (6), we get $A_0 \vdash a_1[v/x] : \tau_1$, from which the conclusion easily follows.

Otherwise, a canonical derivation of (2) ends as:

$$\dfrac{\dfrac{(7)\ A_0 \vdash a_1 : \tau_1}{A_0 \vdash a_1 : \forall\vec{\alpha}_1 \cdot \tau_1}\ (\textsc{Gen}^*) \qquad A_0 \vdash K : \texttt{All}(\vec{\alpha}_1)\,\tau_1 \to \tau}{A \vdash K\ a_1 : \tau}\ (\textsc{All})$$

By induction hypothesis applied to (7) we get $A \vdash a_1[v/x] : \tau_2$. Since $a_1[v/x]$ is at least as generalizable as a_1 we may applied the same Gen rules, and easily conclude.

Case a is `strip` K `of` a_1: There is a canonical derivation of (2) of the form:

$$(8)\ \frac{A_0 \vdash a_1 : \tau_1 \qquad A_0 \vdash K : \mathtt{All}(\vec{\alpha}_2)\,\tau_2 \to \tau_1}{A_0 \vdash \mathtt{strip}\ K\ \mathtt{of}\ a_1 : \tau}\ (\textsc{Strip})$$

where τ is of the form $\tau_2[\tau_i/\alpha_i]$. We conclude by applying the induction hypothesis to (8) and a STRIP rule.

Case a is $a_1\ a_2$ and `fun` $x \to a_1$: Those cases are similar to case STRIP.

Case a is $\{\_\}$: Then a is a value, and variable x is free in a. The conclusion (3) follows from (2) by context extension lemma.

■

Lemma 19 (Existential elimination) *If $A, \Omega_j, x : \forall\alpha_0 \cdot \tau_1[\vec{\Omega}_j(\tau_0))/\vec{\alpha}_j] \vdash a : \tau$, and $\vec{\tau}_j$ are terms whose variables are also variables of τ_0 then the formula $A[x : \forall\vec{\alpha}_0 \cdot \tau_1[\tau_j/\alpha_j]] \vdash a : \tau$ is valid whenever it is well-formed.*

Proof: Let A be a type assignement, Ω_j type symbols that do not appear in A, τ_0 and τ be types whose variables are also variables of τ_0.

We write $\varphi(\tau)$ the term τ where all occurrences of subterms $\Omega_j(\tau_u)$ are sucessivelly replaced (bottom up order always terminates) by $\mu(\tau_j)$ where μ is the smallest substitution such that $\mu(\tau_0)$ is τ_1; $\varphi(\tau)$ is undefined if one of the μ's is.

If $A, \Omega_j, B \vdash a : \tau$ and $\varphi(B)$ and $\varphi(\tau)$ are defined, then $A\varphi(B) \vdash a : \varphi(\tau)$. Remark that φ is well-sorted, compatible with the structure of terms except for Ω_j symbols. Free variables of φ never introduce new variables. The lemma is proved for any context B by structural induction on a.

Case a is x: A canonical derivation ends with a GET rule. Thus τ is an instance of $B(x)$ by a substitution μ. The type $\varphi(\tau)$ equal to $\varphi(\mu(B(x))$, i.e. $(\varphi\circ\mu)(\tau)$, which is equal to $(\varphi\circ\mu)(\varphi(B(x)))$, thus $\varphi(\tau)$ is an instance of $\varphi(B(x))$. The declaration of Ω_j can be removed from the context since no Ω_j occurs in the conclusion.

Case a is `let` $K\ x = a_1$ `in` a_2: A canonical derivation of (1) ends as:

$$\frac{\dfrac{(3)\ A, \vec{\Omega}_j, B \vdash a_1 : \tau_1}{A, \vec{\Omega}_j, B \vdash a_1 : \forall\alpha_1 \cdot \tau_1}\ (\textsc{Gen}^*) \qquad \begin{array}{l} A, \vec{\Omega}_j, B \vdash K : \forall\vec{\alpha}_1, \vec{\alpha}_k \cdot \mathtt{Exist}(\vec{\alpha}_k)\,\tau_2 \to \tau_1\ (4) \\ A, \vec{\Omega}_j, B, y : \forall\vec{\alpha}_1 \cdot \tau_2[\vec{\Omega}_k(\tau_1)/\vec{\alpha}_k] \vdash a_2 : \tau\ (5) \end{array}}{A, \vec{\Omega}_i, B \vdash \mathtt{let}\ K x = a_1\ \mathtt{in}\ a_2 : \tau}$$

By induction hypothesis applied to (3), we get $A, \varphi(B) \vdash a_1 : \varphi(\tau_1)$. We can apply the same GEN rules since $FV(\varphi(B))$ is included in $FV(B)$. More applications of GEN might still be possible since the inclusion is strict in general. We can prove $A, \varphi(B) \vdash K : \forall\vec{\alpha}_1, \vec{\alpha}_j \cdot \mathtt{Exist}(\vec{\alpha}_k)\,\varphi(\tau_2) \to \varphi(\tau_1)$ from (4) reasoning as above in the case where a is x. We apply the indcution hypothesis to (5) and conlcude with a rule LET.

Othercases: They are all similar. The induction hypothesis is applied to one of the premisse of the last rule in a canonical derivation of (1) using compatibility of φ with the structure of terms. Then, the conclusion follows by the same typing rule. ■

Proof: (of redex contraction) It is immediate to check on the reduction rules for each redex that a_2 is generalizable whenever a_1 is generalizable. Let A_1 be a type environment and τ a type such that $A_1 \vdash a_1 : \tau$ (1) and $\Vdash s_1 : A_1$. We show that there exists a store extension A_2 of A_1 such that $A_2 \vdash a_2\tau$ (2) and $\Vdash s_2 : A_1$ by cases on the redex a_1. Moreover, whenever a_1 is generalizable, we shall have A_2 equal to A_1. Each case is shown independently.

Case a_1 is $(\mathtt{fun}\, x \to a)\, v$: A canonical derivation of (1) ends as:

$$\dfrac{\dfrac{\dfrac{(3)\ \ A_1, x : \tau_1 \vdash a : \tau}{A_1 \vdash a : \tau_1 \to \tau}\ (\textsc{Fun})}{A_1 \vdash a : \tau_1' \to \tau'}\ (\textsc{Equal}^*) \qquad (4)\ \ A_1 \vdash v : \tau_1'}{A_1 \vdash a_1 : \tau'}\ (\textsc{App})$$

Rule Equal here is essential. However the only possibility for $\tau_1 \to \tau$ to be equal to $\tau_1' \to \tau'$ is that τ_1 is equal to τ_1' and τ is equal to τ'. Thus we can omit the equaliyt rule and the primes. Note that this would not be the case if we had an axiom such as $(\mathtt{int} \to \mathtt{int}) = (\mathtt{int} \to \mathtt{unit})$.

The previous lemma applied to (4) and (3) shows the conclusion with A_1 for A_2.

Case a_1 is $\mathtt{let}\ K\ x = K\ v\ \mathtt{in}\ a$: A canonical derivation of (1) is

$$\dfrac{\dfrac{\dfrac{(4)\ \ A_1 \vdash v : \tau_2}{A_1 \vdash K : \mathtt{Exist}(\vec{\tau}_j)\, \tau_2 \to \tau_1}\ (\textsc{Exist}) \qquad A_1 \vdash K : \forall \vec{\alpha}_1, \vec{\alpha}_j \cdot \mathtt{Exist}(\vec{\alpha}_j)\, \tau_0 \to \tau_1}{A_1 \vdash K\ v : \tau_1}\ (\textsc{Gen}^*) \qquad (5)\ \ A_1, \vec{\Omega}_j, x : \forall \vec{\alpha}_1 \cdot \tau_0[\vec{\Omega}_j(\tau_1)/\vec{\alpha}_j] \vdash a : \tau}{A_1 \vdash \mathtt{let}\ K\ x = K\ v\ \mathtt{in}\ a : \tau}$$

By existential elimination lemma applied to (5), we deduce

$$A_1, x : \forall \vec{\alpha}_1 \cdot \tau_0[\vec{\tau}_j/\vec{\alpha}_j] \vdash a : \tau.$$

Necessarily, τ_2 must be $\tau_0[\vec{\tau}_j/\vec{\alpha}_j]$, i.e. we have $A_1, x : \forall \vec{\alpha}_1 \cdot \tau_2 \vdash a : \tau$ (6). From (3), we may deduce $A_1 \vdash v \vdash \forall \vec{\alpha}_1 \cdot \tau_2$ (7) by rule Gen. We conclude by applying term substitution to (6) and (7) with A_1 for A_2.

Case a_1 is $\mathtt{strip}\ K\ \mathtt{of}\ Kv$:

$$\dfrac{\dfrac{(8)\ \ A_1 \vdash v : \forall \vec{\alpha} \cdot \tau_0}{A_1 \vdash K : \forall \vec{\alpha} \cdot \mathtt{All}(\vec{\alpha})\, \tau_0 \to \tau_1}\ (\textsc{All}) \qquad A_1 \vdash K : \mathtt{All}(\vec{\tau}_2)\, \tau \to \tau_1}{A_1 \vdash \mathtt{strip}\ K\ \mathtt{of}\ v : \tau}\ (\textsc{Strip})$$

Necessarily, τ must be $\tau_0[\vec{\alpha}/\vec{\alpha}_2]$, thus the conclusion follows by substitution lemma applied to (8) with A_1 for A_2.

Case Extend-S: a_1 is $\{\vec{\ell}_i = \vec{w}_i\} \parallel \{\ell = v\}$ where ℓ is not in ℓ_i. A canonical derivation of (1) ends as:

$$\dfrac{\dfrac{A_1 \vdash \vec{w} : \vec{\tau}}{A_1 \vdash \{\vec{\ell} = \vec{w}\} : \{\vec{\ell} : \vec{\varphi}_i\, \mathtt{pre}.\vec{\tau}\ ;\ \mathtt{abs}\}}\ (\textsc{Record}) \qquad A_1 \vdash v : \tau_0 \qquad (1)}{A_1 \vdash a_1 : \{\ell : \mathtt{static}\, \mathtt{pre}.\tau_0\ ;;\ \vec{\ell}_i : \mathtt{pre}.\vec{\tau}_i\ ;\ \mathtt{abs}\}}\ (\textsc{App})$$

where φ_i is `static` if w is a value and `mut` if w is a store location and the sub-derivation (1) is

$$\frac{A_1 \vdash (\_ \parallel \{\ell = \_\}) : \ldots}{A_1 \vdash (\_ \parallel \{\ell =\}) : \{\ell\colon \mathtt{abs}\ ;\ (\vec{\ell_i}\colon \vec{\varphi}_i\,\mathtt{pre}.\vec{\tau}\ ;\ \mathtt{abs})\} \to \tau_0 \to \{\ell\colon \mathtt{static}\,\mathtt{pre}.\tau_0\ ;\ (\vec{\ell}\colon \vec{\varphi}_i\,\mathtt{pre}.\vec{\tau}\ ;\ \mathtt{abs})\}}\ (\textsc{Get})$$

The equality is also important here (as in all other redexes coming from the application of a primitive). For instance, it would be wrong if for instance $\mathtt{pre}(\tau_{i_0})$ could be changed into `abs`

We can derive:

$$\frac{\dfrac{A_1 \vdash \vec{v}_i : \vec{t}_i \qquad A_1 \vdash v : \tau_0}{A_1 \vdash \{\vec{\ell_i} = \vec{w}_i, \ell = w\} : \{\ell\colon \mathtt{static}\,\mathtt{pre}.\tau_0\ ;\ \vec{\ell_i}\colon \vec{\varphi}_i\,\mathtt{pre}.\vec{\tau}_i\ ;\ \mathtt{abs}\}}\ (2)\ (\textsc{App})}{A_1 \vdash a_2 : \{\ell\colon \mathtt{static}\,\mathtt{pre}.\tau_0\ ;\ \vec{\ell_i}\colon \vec{\varphi}_i\,\mathtt{pre}.\vec{\tau}_i\ ;\ \mathtt{abs}\}}\ (\textsc{Record})$$

where (2) is an instance of the type of the primitive `copy` . This proves the conclusion with A_1 as A_2.

Case Override-S: The case were label ℓ is redefined is similar.

Case Extend-M: The case were label ℓ is mutable starts as above, and then ends with

$$\frac{\dfrac{A_2 \vdash \vec{v}_i : \vec{t}_i \qquad A_2 \vdash l : \tau_0}{A_2 \vdash \{\vec{\ell_i} = \vec{w}_i, \ell = w\} : \{\ell\colon \mathtt{mut}\,\mathtt{pre}.\tau_0\ ;\ \vec{\ell_i}\colon \vec{\varphi}_i\,\mathtt{pre}.\vec{\tau}_i\ ;\ \mathtt{abs}\}}\ (2)\ (\textsc{Copy})}{A_2 \vdash a_2 : \{\ell\colon \mathtt{mut}\,\mathtt{pre}.\tau_0\ ;\ \vec{\ell_i}\colon \vec{\varphi}_i\,\mathtt{pre}.\vec{\tau}_i\ ;\ \mathtt{abs}\}}\ (\textsc{Record})$$

where (2) is an instance of the type of the primitive `copy` and A_2 is $A_1, l \mapsto \tau_0$. We easily check that $A_2 \vdash s, l \mapsto v$, hence the conclusion.

Case Override-M: The case were label ℓ is redefined is similar.

Case Copy: This case is obvious, just taking taking $A_1, \vec{l'_j} \mapsto A_1(\vec{l_j})$ for A_2.

Case Dot-S: a_1 is $\{\vec{l_i} = \vec{w}_i\ ;\ \ell = v\}.\ell$. A canonical derivation of (1) ends as:

$$\frac{A \vdash w_i : \tau_i \qquad A \vdash v : \tau\ (1) \qquad A \vdash (\_.\ell) : \{\ell\colon \mathtt{static}\,\mathtt{pre}.\tau\ ;\ (\vec{\ell_i}\colon \varphi_i\,\mathtt{pre}.\vec{\tau}_i\ ;\ \mathtt{abs})\} \to \tau}{A \vdash a_1 : \tau}\ (\textsc{App})$$

Form (1), the conclusion holds for A_1 as A_2.

Case Dot-M: As above, but v is l in (1) and static is replaced by `mut` in the right premise. Hence l is in the domain of s and $A_1 \vdash s(l) : \tau$, thus the conclusion holds for A_1 as A_2.

Case Mute: As above, one easily shows that l is in the domain of A_1 and s and that both $A_1 \vdash v : \tau$ and $A_1 \vdash s(l) : \tau$ holds. Hence $A_1 \vdash s, l \mapsto v$ and the conclusion holds for A_1 as A_2.

■

The second theorem asserts that well-typed terms that cannot be reduced are values, thus the evaluation is never "stuck." It is proved by structural induction on the value using the following lemma, which is itself a consequence of syntacticness of the theory 5.5.

Lemma 20 *Let A be a store extension of A_0 such that $A \vdash v : \tau$.*

- *if τ is a functional type then v is a function or a constant.*

- *if τ is a record type then v is a record; moreover, if τ is of the shape $\{\ell\colon \mathtt{static}\,\mathtt{pre}.\tau_1\ ;\ \mathtt{static}\tau_2\}$, field ℓ is defined and $v(\ell)$ is a value. if τ is of the shape $\{\ell\colon \mathtt{mut}\,\mathtt{pre}.\tau_1\ ;\ \mathtt{static}\tau_2\}$, field ℓ is defined and $v(\ell)$ is a location.*
- *if τ is $k(\tau_1)$ then v is a value Kv_1 where K and k are paired in A.*

Proof: We show separately:

- If v is a function or a constant then τ is a functional type.
- If v is a record then τ is a record type.

  Moreover, if $v(\ell)$ is defined and is a value, then then τ is of the shape $\{\ell\colon \mathtt{static}\,\mathtt{pre}.\alpha_1\ ;\ \alpha_2\}$;
  Moreover, if $v(\ell)$ is defined and is a location, then then τ is of the shape $\{\ell\colon \mathtt{mut}\,\mathtt{pre}.\alpha_1\ ;\ \alpha_2\}$;
- If a is a value Kv_1 then τ is a $k(\tau_1)$ where K and k are paired.

It is immediate to show that τ has the right shape modulo type equality. The type-consistency property shows that equality cannot change the top structure of types, thus the set of types are disjoint. Since all cases of values have been considered, this proves the lemma. ■

Proof: (of normal forms) The proof is by structural induction on a. Let A be a store extension of the initial environment such that $A \vdash a : \tau$ (1) for some type τ and $\Vdash s : A$ for some store s. We assume that a is a value or a/s can be reduced.

Case a is x: Since A only assign types to constants and locations x must be a constant, therefore it is a value.

Case a is $a_1\ a_2$: A canonical derivation of (1) shows that there exists a type τ_1 such that $A \vdash a_1 : \tau_1 \to \tau$ and $A \vdash a_2 : \tau_2$. The induction hypothesis shows that either a_1 or a_2 can be reduced, or both are values. In the former case, a can be reduced. In the later case, since a has a functional type, either it is a function $\mathtt{fun}\ x \to a_0$ and a can be reduced by rule FUN or it is a primitive c. Then, reasoning by case on the primitive c, we can easily show in each case that the shape of the value a_2 is such that one reduction rule always applies.

Case a is $\mathtt{let}\ KKx = b\ \mathtt{in}\ a_2$: A canonical derivation of (1) ends as:

$$\dfrac{\dfrac{(2)\ A \vdash b : \tau_1}{A \vdash b : \forall\alpha_1 \cdot \tau_1}\ (\text{GEN}^*) \qquad A \vdash K : \forall\vec{\alpha}_1, \vec{\alpha}_j \cdot \mathtt{Exist}(\vec{\alpha}_j)\,\tau_0 \to \tau_1 \qquad \ldots}{A \vdash \mathtt{let}\ K\ x = b\ \mathtt{in}\ a_2 : \tau_2}\ (\text{LET})$$

The induction hypothesis applied to (2) shows that b is a value. Since it has a K type, it must be a value Kv. Then a could be further reduced.

Case a is $\mathtt{strip}\ K\ \mathtt{of}\ a_1$: Easy.

Case a is $K\ a_1$: Easy.

Othercases: If a is c, $\mathtt{fun}\ x \to a$, or $\{\vec{\ell} = \vec{w}\}$, then it is a value.

■

Chapter 6

Objective ML: Une extension de ML avec des objets primitifs

Ce chapitre, publié dans [113], est le résultat d'un travail en collaboration avec Jérôme Vouillon.

Objective ML :
Une extension de ML avec des objets primitifs

Objective ML est une petite extension de ML avec des objets primitifs et des classes au niveau supérieur du langage. Elle est complètement compatible avec ML ; son système de type s'appuie sur le polymorphisme de ML, les types enregistrements et un meilleur traitement des abréviations de type. Objective ML offre la plupart des fonctionnalités des langages à objets, ce qui inclue l'héritage multiple, la possibilité pour une méthode de retourner l'objet lui-même, les méthodes binaires, ainsi que les classes paramétriques. Cela montre que les objets peuvent être ajoutés aux langages typés avec inférence des types fondés sur le polymorphisme de ML.

Objective ML: an extension of ML with primitive objects

Objective ML is a small practical extension to ML with objects and toplevel classes. It is fully compatible with ML; its type system is based on ML polymorphism, record types with polymorphic access, and a better treatment of type abbreviations. Objective ML allows for most features of object-oriented languages including multiple inheritance, methods returning self and binary methods as well as parametric classes. This demonstrates that objects can be added to strongly typed languages based on ML polymorphism.

Introduction

We propose a simple extension to ML with class-based objects. *Objective ML* is a fully conservative extension to ML. A beginner may ignore the object extension. Moreover, he would not notice any difference, even in the types inferred. This is possible since the type inference algorithm of Objective ML, as in ML, is based on first-order unification and let-binding polymorphism. Types are extended with object types that are similar to record types for polymorphic access. Both the status and the treatment of type abbreviations have been improved in order to keep types readable.

When using object-oriented features, the user is never required to write interfaces of classes, although he might have to include a few type annotations when defining parametric classes or coercing objects to their counterparts in super classes.

Objective ML is a class-based system. Objects are records of methods. Our language copes with most features of object-oriented programming, including methods returning self, binary methods, virtual classes and multiple inheritance. Coercion from objects to their counterparts in super classes is also possible.

The ingredients used, except automatic abbreviations, are not new. However, their incorporation into a practical language, combining power, simplicity and compatibility with ML, is new.

Objective ML is formally defined, and its dynamic semantics is proven correct with respect to the static semantics. The language has not been designed to be a minimal calculus of objects, but rather the core of a real programming language. In particular, the semantics of classes is compatible with programming in imperative style as well as in functional style and it allows for efficient memory management (methods can be shared between all the instances of a class).

This paper is organized as follows: the first section is an overview of Objective ML. Objects and classes are introduced in sections 6.2 and 6.3. Coercions are dealt with in section 6.4. The semantics of the language is described in section 6.5. Type inference is discussed in section 6.6. The abbreviation mechanism is explained in sections 6.7 and 6.8. Extensions to the core language are presented in sections 6.9 and 6.10. In section 6.11, we compare our proposal with other works.

6.1 An overview of Objective ML

Objective ML has been implemented on top of the Caml Special Light system [70]. We have used this implementation, now called Objective Caml[1], to process all examples shown below. When useful, we display the output of the typechecker in a slanted font. Toplevel definitions are implicit `let .. in ...` For each phrase, the typechecker outputs the binding that will be generalized and added to the global environment before starting to typecheck the next phrase.

The language Objective ML is class-based. That is, objects are usually created from classes, even though it is also possible to create them directly (this is described in the next section). Here is a straightforward example of a class `point`.

```
class point x0 = struct
  field x = ref x0
  method move d = (x := !x + d; !x)
end;;
class point : int → sig
  field x : int ref
  method move : int → int
```

[1]The syntax has been slightly modified here in order to keep the concrete syntax and the abstract syntax closer.

```
end
```

Class types are automatically inferred. Objects are usually created as instances of classes. All objects of the same class have the same type structure, reflecting the structure of the class. It is important to name object types to avoid repeating the whole nested, often recursive, structure of objects at each occurrence of an object type. Thus, the above declaration also automatically defines the abbreviation:

```
type point = ⟨move : int → int⟩
```

which is the type of objects with a method `move` of type `int → int`. In practice, this is essential in order to report readable types to the user. The following example shows that these object abbreviations are introduced when the operator `new` is applied to a class.

```
new point;;
- : int → point = ⟨fun⟩
let p = new point 3;;
value p : point = ⟨obj⟩
```

Classes can also be derived from other classes by adding fields and methods. The following example shows how an object sends messages to itself; for instance, if the `scale` formula is overridden in a subclass, the `move` method will use the new scale. Here, methods of the parent class are bound to the super-class variable `parent` and are used in the redefinition of the `move` method (the binary operator # denotes method invocation in Objective ML).

```
class scaled_point s0 = struct
  inherit point 0 as parent
  field  s = s0
  method scale = s
  method move d =
    parent#move (d * self#scale)
end;;
class scaled_point : int → sig
  field s : int
  field x : int ref
  method move : int → int
  method scale : int
end
```

Scaled points have a richer interface than points. It is still possible to consider scaled points as points. This might be useful if one wants to mix different kinds of points with incompatible attributes, ignoring anything but the interface of points:

```
let points = [(new scaled_point 2 : scaled_point ⟨: point); new point 1];;
value points : point list = [⟨obj⟩; ⟨obj⟩]
```

A few other examples are given in the paper, and an example using binary methods can be found in the appendix 6.2.

Notation

A *binding* is a pair (k, t) of a key k and an element t. It is written $k = t$ when t is a term or $k : t$ when t is a type. Bindings may also be tagged. For instance, if `foo` is a tag, we write `foo` $u = a$ or `foo` $u : a$. Tags are always redundant in bindings and are only used to remind what kind of identifier is bound.

Term sequences may contain several bindings of the same key. We write @ for the concatenation of sequences (i.e. their juxtaposition). On the contrary, linear sequences cannot bind the same key several times. We write + for the overriding extension of a sequence with another one, and $\oplus$ to enforce that the two sequences must be compatible (i.e. they must agree on the intersection of their domains). We write $\emptyset$ for the empty sequence.

A *sequence* can be used as a function. More precisely, the *domain* of a sequence S is the union, written $dom\,(S)$, of the first projection of the elements of the sequence. An element of the domain k is mapped to the value t so that $x : t$ is the rightmost element of the sequence whose first projection is x, ignoring the tags. The sequence $S \setminus$ `foo` is composed of all elements of S but those tagged with `foo`. Finally, we write `foo` (S) for $\{k : t \mid$ `foo` $k : t \in S\}$, that is, for the subsequence of the elements of S tagged with `foo` but stripped of the tag `foo`.

We write $\bar{t}$ for a tuple of elements $(t_i{}^{i \in I})$ when indexes are implicit from the context.

6.2 Objects

We assume that a set of variables $x \in \mathcal{X}$ and two sets of names $u \in \mathcal{U}$ and $m \in \mathcal{M}$ are given. Variable x is used to abstract other expressions; x is bound in `fun` (x) a and `let` $x = a_1$ `in` a_2. Programs are considered equal modulo renaming of bound variables. Conversely, names are always free and not subject to α-conversion: u and m are used to name field and method components of objects, respectively. The syntax of expressions is provided below.

$$
\begin{array}{rl}
a ::= & x \mid \texttt{fun}\ (x)\ a \mid a\ a \mid \texttt{let}\ x = a\ \texttt{in}\ a \\
& \mid \texttt{self} \mid u \mid \{\langle u = a; \ldots\ u = a\rangle\} \mid a\#m \\
& \mid \langle \texttt{field}\ u = a\,; \ldots \texttt{field}\ u = a\,; \texttt{method}\ m = a\,; \ldots \texttt{method}\ m = a\rangle
\end{array}
$$

Operations on references could be included as constants k (the ellipsis in syntax definitions means that we are extending the previous definition; "_" marks the positions of arguments around prefix or infix constants):

$$a ::= \ldots \mid k \qquad \text{and} \qquad k ::= \texttt{ref}\ \_ \mid (\_ := \_) \mid (!\_)$$

For the sake of simplicity, we omit them in the formalization, although they are used in the examples. An object is composed of a sequence of *field* bindings—the hidden internal state—, and of a sequence of *method* bindings for accessing and modifying these fields. Fields are also called *instance variables*. The type of an object is thus the type of the record of its methods. In an object, a method may return the object itself or expect to be applied to another object of the same kind. Types might thus be recursive. We assume given two countable collections of type variables and row variables,

written α and ρ, and a collection of type constructors written κ.

$$\begin{array}{lr} \tau ::= \alpha \mid \tau \to \tau \mid (\tau, \ldots \tau)\,\kappa \mid \mathtt{rec}\ \alpha.\tau \mid \langle \tilde{\tau} \rangle & \text{Types} \\ \tilde{\tau} ::= (m : \tau; \tilde{\tau}) \mid \rho \mid \emptyset & \text{Object types} \\ \sigma ::= \forall\, \bar{\alpha}.\, \tau & \text{Type schemes} \end{array}$$

Object types ending with a row variable are named *open object types*, while others are named *closed object types*. In the examples, closed object types are simply written $\langle {m_i : \tau_i}^{i \in I} \rangle$, i.e. the $\emptyset$ symbol is omitted. The row variables of open object types are also left implicit in an ellipsis $\langle {m_i : \tau_i}^{i \in I}; .. \rangle$ (abbreviations explained in section 6.8 can even be used to share ellipsis). In the formal presentation, we keep both $\emptyset$ and row variables explicit. A label can only appear once in an object type. This is easily ensured by sorting type expressions [104]. The distinction between τ and $\tilde{\tau}$ can also be guaranteed by sorts. Thus, we omit the distinction and simply write τ below.

Type equality is defined by the following family of left-commutativity axioms:

$$(m_1 : \tau_1; m_2 : \tau_2; \tau) = (m_2 : \tau_2; m_1 : \tau_1; \tau)$$

plus standard rules for recursive types [6]:

$$\frac{\tau_1 = \tau_2}{\mathtt{rec}\ \alpha.\tau_1 = \mathtt{rec}\ \alpha.\tau_2}\ (\textsc{Rec}) \qquad \mathtt{rec}\ \alpha.\tau = \tau[\mathtt{rec}\ \alpha.\tau/\alpha]\ (\textsc{Fold-Unfold})$$

$$\frac{\tau_1 = \tau[\tau_1/\alpha] \wedge \tau_2 = \tau[\tau_2/\alpha] \qquad \mathtt{rec}\ \alpha.\tau \text{ well-formed}}{\tau_1 = \tau_2}\ (\textsc{Contract})$$

Recursive types $\mathtt{rec}\ \alpha.\tau$ are only well-formed if τ is neither a variable nor of the form $\mathtt{rec}\ \alpha'.\tau'$ (this is not too restrictive since $\mathtt{rec}\ \alpha.(\mathtt{rec}\ \alpha'.\tau')$ can always be rewritten $\mathtt{rec}\ \alpha.\tau'[\alpha/\alpha']$). This guarantees that τ is contractive in α, and ensures that $\mathtt{rec}\ \alpha.\tau$ effectively defines a regular tree. Types, sorts, and type equality are a simplification of those used in [108], which we refer to for details. Typing contexts are sequences of bindings:

$$A ::= \emptyset \mid A + x : \sigma \mid A + \mathtt{field}\ u : \tau \mid A + \mathtt{self} : \tau$$

Typing judgments are of the form $A \vdash a : \tau$. The typing rules for ML are recalled in appendix 6.1.

Typing rules for objects are given in figure 6.1. A simple object is just a set of methods. Methods can send messages to the object itself, which will be bound to the special variable `self`. A simple object could be typed as follows:

$$\frac{A + \mathtt{self} : \langle {m_j : \tau_j}^{j \in J} \rangle \vdash {a_j : \tau_j}^{j \in J}}{A \vdash \langle \mathtt{method}\ {m_j = a_j}^{j \in J} \rangle : \langle {m_j : \tau_j}^{j \in J} \rangle}$$

However, an object can also have instance variables. Instance variables may only be used inside methods defined in the same object. The typechecking of instance variables $(\mathtt{field}\ u_i = a_i)^{i \in I}$ of an object produces a typing environment $(\mathtt{field}\ u_i : \tau_i)^{i \in I}$ in which the methods are typed (rules Object and Field).

Instance variables also provide the ability to clone an object possibly overriding some of its instance variables (rule Override). In this rule, types τ_y and τ_i do not seem to be connected.

$$\frac{\texttt{field}\ u : \tau \in A}{A \vdash u : \tau}\ (\textsc{Field}) \qquad \frac{(\texttt{field}\ u_i : \tau_i \in A \qquad A \vdash a_i : \tau_i)^{i \in I} \qquad \texttt{self} : \tau_y \in A}{A \vdash \{\langle u_i = {a_i}^{i \in I} \rangle\} : \tau_y}\ (\textsc{Override})$$

$$\frac{A^\star \vdash {a_i}^{i \in I} : {\tau_i}^{i \in I} \qquad A^\star + \texttt{self} : \langle {m_j : {\tau_j}^{j \in J}} \rangle + \texttt{field}\ {u_i : {\tau_i}^{i \in I}} \vdash {a_j : {\tau_j}^{j \in J}}}{A \vdash \langle \texttt{field}\ {u_i = {a_i}^{i \in I}}\ ; \texttt{method}\ {m_j = {a_j}^{j \in J}} \rangle : \langle {m_j : {\tau_j}^{j \in J}} \rangle}\ (\textsc{Object})$$

(This rule will be overridden by the more general rule of same name in figure 6.3.)

$$\frac{A \vdash a : \langle m : \tau ; \tau' \rangle}{A \vdash a \# m : \tau}\ (\textsc{Send})$$

Figure 6.1: Typing rules for objects

They are however, thanks to typing rule OBJECT which requires the type τ_y of `self` and the types τ_i of instance variables to be related to the same object. This is also ensured by typing the premises in the context $A^\star$ equal to $A \setminus \{\texttt{field}, \texttt{self}\}$. As a result, the expression $\langle \texttt{field}\ u = a\ ; \texttt{method}\ m = \langle \texttt{method}\ m = u \rangle \rangle$ becomes ill-typed. This is not a real restriction however, since one can still write the less ambiguous expression $\langle \texttt{field}\ u = a\ ; \texttt{method}\ m = \texttt{let}\ x = u\ \texttt{in}\ \langle \texttt{method}\ m = x \rangle \rangle$.

The rule SEND for method invocation is similar to the rule for polymorphic access in records: when sending a message m to an object a, the type of a must be an object type with method m of type τ; the object may have other methods that are captured in the row expression τ'. The type returned by the invocation of the method is τ. The type of method invocation may also be seen below:

```
let send_m a = a#m;;
value send_m : < m : 'a; .. > -> 'a = <fun>
```

The ellipsis stands for an anonymous row variable ρ, which means that any other method than m may also be defined in the object a. Row variables provide parametric polymorphism for method invocation. Instead of using row variables, many other languages use subtyping polymorphism. Since subtyping polymorphism must be explicitly declared in Objective ML (see section 6.4), row variables are essential here to keep type inference. Row variables also allow to express some kind of matching [16] without F-bounded or higher-order quantification [93, 2, 3]. Here is an example:

```
let min x y = if x#leq y then x else y;;
value min :
  (< leq : 'a -> bool; .. > as 'a) ->
  'a -> 'a = <fun>
```

The binder "as" makes it possible to deal with open object types occurring several times in a type (this will be detailed in section 6.8). An expanded version of this type is:

$$\texttt{rec}\ \alpha.\langle leq : \alpha \to bool; \rho \rangle \to \texttt{rec}\ \alpha.\langle leq : \alpha \to bool; \rho \rangle \to \texttt{rec}\ \alpha.\langle leq : \alpha \to bool; \rho \rangle$$

The function `min` can be used for any object of type τ with a method $leq : \tau \to bool$, since the row variable ρ can always be instantiated to the remaining methods of type τ.

$$
\begin{array}{rll}
a & ::= \ldots \mid \langle b \rangle \mid \texttt{class}\ z = c\ \texttt{in}\ a \mid \texttt{new}\ c \mid s\#m & \text{Expressions} \\
c & ::= z \mid \texttt{fun}\ (x)\ c \mid \texttt{struct}\ b\ \texttt{end} & \text{Class expressions} \\
b & ::= \emptyset \mid d\,;\,b & \text{Class bodies} \\
d & ::= \texttt{inherit}\ c\ \texttt{as}\ s \mid \texttt{field}\ u = a \mid \texttt{method}\ m = a &
\end{array}
$$

Figure 6.2: Core class syntax

6.3 Classes

The syntax for classes, introduced in section 6.1, is formally given in figure 6.2. The body of a class is a sequence b of small definitions d. We assume as given a collection of class identifiers $z \in \mathcal{Z}$, and a collection of super-class identifiers written s.

We have also enriched the syntax of objects so that it reflects the syntax of classes. That is, objects can also be built using inheritance, and fields need not precede methods.

In practice, classes will only appear at the toplevel. However, it is simpler to leave more freedom, and let them appear anywhere except under abstraction. Technically, it would be possible to make them first-order, that is to allow abstraction of classes; however, class types should be provided explicitly in abstractions. The little gain in practice is probably not worth the complication (a class can be already parameterized by other classes using modules).

The type of a class structure, $\texttt{sig}\ (\tau_y)\ \varphi\ \texttt{end}$, is composed of the type τ_y of self (i.e. the type an object of this class would have), and the type φ of its field bindings and method bindings. Class types are written γ. Type schemes are extended with class types.

In the concrete syntax, τ_y and φ are combined: methods that appear in τ_y but not is φ are flagged *virtual* (as they are not defined); other methods appear both in τ_y and φ, with the same type. When necessary, a type variable can also be bound to τ_y. For instance, the concrete syntax

$$\texttt{sig (}\alpha\texttt{) virtual copy : }\alpha\texttt{ method x : int end}$$

expands to

$$
\begin{array}{l}
\texttt{sig}\ (\texttt{rec}\ \alpha.\langle \texttt{copy} : \alpha; \texttt{getx} : \texttt{int}; \rho\rangle) \\
\quad \texttt{method getx} : \texttt{int} \\
\texttt{end}.
\end{array}
$$

$$
\begin{array}{rl}
\gamma & ::= \texttt{sig}\ (\tau)\ \varphi\ \texttt{end} \mid \tau \to \gamma \\
\varphi & ::= \emptyset \mid \varphi\,;\texttt{field}\ u : \tau \mid \varphi\,;\texttt{method}\ m : \tau \\
 & \quad \mid \varphi\,;\texttt{super}\ s : \varphi \\
\sigma & ::= \ldots \mid \forall\,\bar{\alpha}.\,\gamma
\end{array}
$$

Typing contexts are extended with class variable bindings and superclass bindings:

$$A ::= \ldots \mid A + z : \sigma \mid A + \texttt{super}\ s : \varphi$$

We add new typing judgments $A \vdash b : \varphi$ and $A \vdash d : \varphi$ that are used to type class bodies. We also redefine $A^\star$ to be A where all `field`, `method`, `super`, and `self` bindings have been removed.

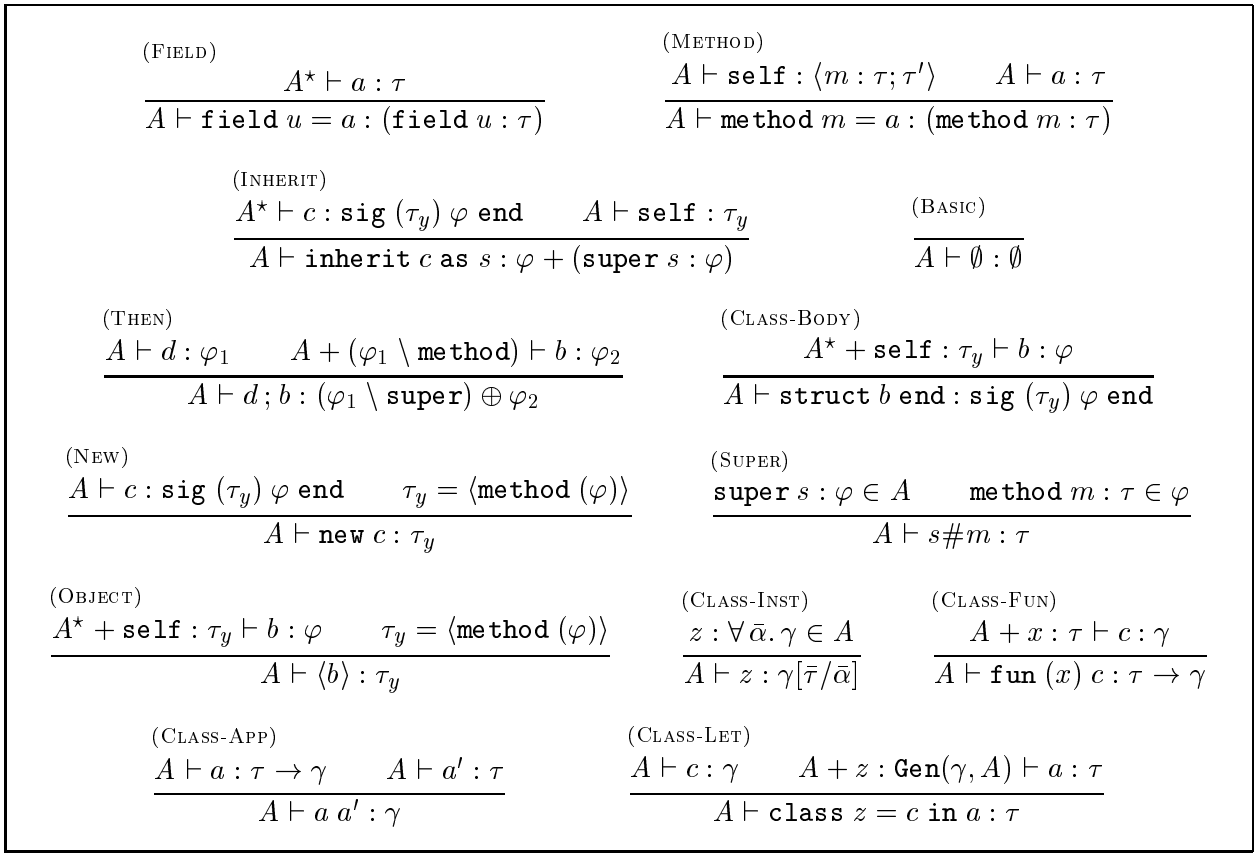


Figure 6.3: Typing rules for classes

Typing rules are given in figure 6.3. Generalization of class types $\texttt{Gen}(\gamma, A)$ is, as for regular types, $\forall\,\bar{\alpha}.\,\gamma$ where $\bar{\alpha}$ are all variables of γ that are not free in A.

Class bodies are typed by adding each component (inheritance clause, field, or method) one after the other. Fields are typed in $A^\star$, so that they cannot depend on other fields (rule FIELD). On the contrary, methods may depend on all fields and super-classes that were previously defined (rule METHOD). The INHERIT rule ensures that `self` is assigned the same type in both the superclass and the subclass; all bindings of the superclass are discharged in the subclass, and the superclass variable is given the type of the superclass. Superclass variables are only visible while typechecking the body of the class but are not exported in the type of the class itself, as shown by rule THEN. The rule OBJECT is more general than (and overrides) the one of figure 6.1; it corresponds to the combination of rule CLASS-BODY and rule NEW.

When a value or method component is redefined, its type cannot be changed, since previously defined methods might have assumed the old type[2]. This is enforced by using in rule THEN the $\oplus$ operator which requires that the two argument sequences be compatible on the intersection of their domains. At first, this looks fairly restrictive. But it still leaves enough freedom in practice. Indeed, the class type can also be specialized by instantiating some type variables. Methods returning objects of the same type as self are thus correctly typed.

```
class duplicable () = struct
  method copy = {< >}
end;;
class duplicable : unit → sig (α)
  method copy : α
end
```

In this class type, α is bound to the type of self. Thus, objects of any subclass of this class have types that match $\texttt{rec}\ \alpha.\langle \texttt{copy} : \alpha;\ ..\rangle$. Class `duplicable` can then be inherited, and method `copy` still have the expected type (that is, the type of self).

```
class duplicable_point x = struct
  inherit duplicable () inherit point x
end;;
class duplicable_point : int → sig (α)
  field x : int ref
  method copy : α
  method move : int → int
end
```

Note that ancestors are ordered, which disambiguates possible method redefinitions: the final method body is the one inherited from the ancestor appearing last.

Rule CLASS-LET, CLASS-INST, CLASS-FUN and CLASS-APP are similar to the rules LET, INST, FUN and APP for core ML (described in appendix 6.1). The two rules CLASS-LET and CLASS-INST are essential since polymorphism of class types enables method specialization during inheritance, as explained above.

[2]One may imagine to relax this constraint, and allow the type of the redefined method to be a subtype of the original method. One would, however, lose a property we believe important: rule INHERIT shows that the type a class gives to self is a common instance of the different types of self in its ancestors; as a consequence, the type of self in a class unifies with the type of any object of a subclass of this class.

6.4 Coercion

Polymorphism on row variables enables to write a parametric function that sends a message m to any object that has a method m. Thus, subtyping polymorphism is not required here. This is important since subtyping is not inferred in Objective ML.

There is still a notion of explicit subtyping, that allows explicit coercion of an expression of type τ_1 to an expression of type τ_2 whenever τ_1 is a subtype of τ_2. As shown in the last example of section 6.1, this enables to see all kinds of points just as simple points, and put them in the same data-structure.

The language of expressions is extended with the following construct:

$$a ::= \ldots \mid (a : \tau <: \tau)$$

The corresponding typing rule is:

$$\frac{\tau \leq \tau' \qquad A \vdash a : \theta(\tau)}{A \vdash (a : \tau <: \tau') : \theta(\tau')} \text{(Coerce)} \qquad \theta \text{ substitution}$$

The premise $\tau \leq \tau'$ means that τ is a subtype of τ'. As long as typechecking is concerned, we could have equivalently introduced coercions as a family of constants $(\_ : \tau <: \tau')$ of respective principal types $\forall\, \bar{\alpha}.\, \tau \to \tau'$ where $\bar{\alpha}$ are free variables of τ and τ' indexed by all pairs of types (τ, τ') such that $\tau \leq \tau'$.

The subtyping relation $\leq$ is standard [6]. We choose the simpler (and algorithmically more efficient) presentation of [63]. The constraint $\tau \leq \tau'$ is defined on regular trees as the smallest transitive relation that obeys the following rules:

Closure rules

$$\begin{gathered}
\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' \Longrightarrow \tau_1' \leq \tau_1 \wedge \tau_2 \leq \tau_2' \\
\langle \tau \rangle \leq \langle \tau' \rangle \Longrightarrow \tau \leq \tau' \\
(m : \tau_1; \tau_2) \leq (m : \tau_1'; \tau_2') \Longrightarrow \tau_1 \leq \tau_1' \wedge \tau_2 \leq \tau_2'
\end{gathered}$$

Consistency rules

$$\begin{gathered}
\tau \leq \tau_1 \to \tau_2 \Longrightarrow \tau \text{ is of the shape } \tau_1' \to \tau_2' \\
\tau \leq \langle \tau_0 \rangle \Longrightarrow \tau \text{ is of the shape } \langle \tau_0' \rangle \\
\tau \leq (m : \tau_1; \tau_2) \Longrightarrow \tau \text{ is of the shape } (m : \tau_1'; \tau_2') \\
\tau \leq \emptyset \Longrightarrow \tau = \emptyset \\
\tau \leq \alpha \Longrightarrow \tau = \alpha,
\end{gathered}$$

Our subtyping relation does not enhance subtyping assumptions on variables, and it is thus weaker than the subtyping relation used in [41], except on ground types.

For instance, the expression `fun` (x) x has type $\forall\, \alpha, \alpha' \mid \alpha \leq \alpha'.\, \alpha \to \alpha'$ in [41], while we can only type the equivalent expression `fun` (x) $(x : \tau <: \tau')$ for particular instances (τ, τ') of (α, α') such that $\tau \leq \tau'$.

Values

$$
\begin{array}{rll}
v & ::= & \ldots \mid \texttt{fun}\ (x)\ a \mid \langle w \rangle \\
v_c & ::= & \texttt{fun}\ (x)\ c \mid \texttt{struct}\ w\ \texttt{end} \\
w & ::= & \emptyset \mid w_d\,;\, w \qquad\qquad \texttt{field} \text{ preceed } \texttt{method}\text{, no overriding} \\
w_d & ::= & \texttt{method}\ m = a \mid \texttt{field}\ u = v
\end{array}
$$

Evaluation contexts

$$
\begin{array}{rll}
E & ::= & [] \mid \texttt{let}\ x = E\ \texttt{in}\ d \mid E\ a \mid v\ E \mid E\#m \mid \langle F \rangle \mid \texttt{new}\ E \mid \texttt{class}\ z = E_c\ \texttt{in}\ d \\
E_c & ::= & [] \mid E_c\ a \mid v_c\ E \mid \texttt{struct}\ F\ \texttt{end} \\
F & ::= & [] \mid F_d\,;\, b \mid w_d\,;\, F \\
F_d & ::= & \texttt{inherit}\ E_c\ \texttt{as}\ s \mid \texttt{field}\ u = E
\end{array}
$$

From classes to objects

$$\texttt{new}\ (\texttt{struct}\ w\ \texttt{end}) \longrightarrow \langle w \rangle$$

Reduction of objects

$$
\begin{array}{rll}
\texttt{inherit}\ (\texttt{struct}\ w\ \texttt{end})\ \texttt{as}\ s\,;\, b & \longrightarrow & w \mathbin{@} (b\,[w(m)/s\#m]^{m \in dom\,(w)}) \\
\texttt{field}\ u = v\,;\, w & \longrightarrow & w \qquad \text{if } u \in dom\,(w) \\
\texttt{method}\ m = a\,;\, w & \longrightarrow & w \qquad \text{if } m \in dom\,(w) \\
\texttt{method}\ m = a\,;\, (\texttt{field}\ u = v\,;\, w) & \longrightarrow & \texttt{field}\ u = v\,;\, (\texttt{method}\ m = a\,;\, w)
\end{array}
$$

Reduction of method invocation ($U = dom\,(w)$)

$$
\begin{array}{rl}
\langle w \rangle \# m \longrightarrow & w(m)[\langle w \rangle / \texttt{self}][w(u)/u]^{u \in U} \\
& [\langle w \mathbin{@} (\texttt{field}\ u = {a_u}^{u \in V}) \rangle / \{\langle u = {a_u}^{u \in V} \rangle\}]^{V \subset U}
\end{array}
$$

Reduction of coercions

$$(a : \tau <: \tau') \longrightarrow a$$

Reduction of other expressions

$$
\begin{array}{c}
\texttt{let}\ x = v\ \texttt{in}\ a \longrightarrow a[v/x] \qquad \texttt{class}\ z = v\ \texttt{in}\ a \longrightarrow a[v/z] \\
(\texttt{fun}\ (x)\ a)\ v \longrightarrow a[v/x] \quad (\texttt{fun}\ (x)\ c)\ v \longrightarrow c[v/x]
\end{array}
$$

Context reduction

$$
\begin{array}{c}
E[a] \longrightarrow E[a'] \text{ if } a \longrightarrow a' \quad E[b] \longrightarrow E[b'] \text{ if } b \longrightarrow b' \\
E[c] \longrightarrow E[c'] \text{ if } c \longrightarrow c'
\end{array}
$$

Figure 6.4: Semantics of Objective ML

6.5 Semantics

We give a small step reduction semantics to our language. Values are of two kinds: regular expression values are either functions or object values. Class values are either class functions or reduced class structures. Object values and reduced class structures are composed of methods and fields which are themselves values; fields must precede methods and neither can be overridden in values. Values, evaluation contexts, and reduction rules are given in figure 6.4.

The first reduction rule shows that objects are just a restricted view of classes where instance variables have been hidden.

We have chosen to reduce inheritance in objects rather than classes. It would also be possible to reduce inheritance inside classes, and reorder methods and fields as well. Our choice is simpler and more general, since classes can also be inherited in objects.

The reduction of object expressions to values is performed in two steps, described by the four rules for objects: inheritance and evaluation of value components are reduced top-down (first rule, we remind that the meta-notation @ stands for the concatenation of sequences); the components are then re-ordered (last rule) and redundant components removed bottom-up (two middle rules).

The invocation of a method $\langle w \rangle \# m$ evaluates the corresponding expression $w(m)$ after replacing self, instance variables, and overriding by their current values. That is, the following substitutions are successively applied:

1. $[\langle w \rangle / \texttt{self}]$ replaces `self` by $\langle w \rangle$,

2. $[w(u)/u]^{u \in dom\,(w)}$ replaces each outer instance variable u by its actual value. Inner instances of u, *i.e.* those appearing inside an object $\langle w' \rangle$, are not replaced since they are related to the inner object. Note that $w(u)$ is a value and does not contain free fields.

3. $[\langle w @ (\texttt{field}\ u = {a_u}^{u \in V}) \rangle / \{\langle u = {a_u}^{u \in V} \rangle\}]^{V \subset U}$ replaces each outer occurrence of an overriding $\{\langle u = {a_u}^{u \in V} \rangle\}$ by a new object built from w by overriding fields $u \in V$ by $(\texttt{field}\ u = a_u)^{u \in V}$. Inner occurrences, *i.e.* those appearing inside an object $\langle w' \rangle$, are not replaced since they are related to the inner object. Note that a_u is non necessarily a value, and may contain other outer overriding of fields, that should be replaced simultaneously, or equivalently in a bottom-up fashion (deeper occurrences being replaced first).

Coercion behaves as the identity function: the coercion of a value reduces to the value itself. Subject reduction can then only be proved by extending the type system with an implicit subtyping rule:

$$\frac{A \vdash a : \tau \qquad \tau \leq \tau'}{A \vdash a : \tau'}\ (\textsc{Sub})$$

This means that a well-typed expression that has been reduced may not always be typable without rule Sub. This is not surprising since explicit subtyping may disappear during reduction. Thus, implicit subtyping may be required after reduction. It is possible however to keep explicit subtyping information during reduction, and avoid the need for rule Sub. This would be obtained by replacing the rule

$$(a : \tau <: \tau') \longrightarrow a$$

by the following rules

$$(v : \langle {m_i : \tau_i}^{i \in I} \rangle <: \langle {m_i : \tau'_i}^{i \in J} \rangle)$$

$$\text{(Fuse)}\quad \frac{\alpha \doteq e \wedge \alpha \doteq e'}{\alpha \doteq e \doteq e'} \qquad \text{(Decompose) (1)}\quad \frac{f(\alpha_i{}^{i\in I}) \doteq f(\alpha'_i{}^{i\in I}) \doteq e}{f(\alpha_i{}^{i\in I}) \doteq e \wedge (\alpha_i \doteq \alpha'_i)^{i\in I}} \qquad \text{(Generalize) (2)}\quad \frac{e[\tau/\alpha] \qquad \alpha \notin \tau}{\exists\, \alpha.\, e \wedge \alpha \doteq \tau}$$

$$\text{(Mutate)}\quad \frac{(m_1 : \alpha_1; \alpha'_1) \doteq (m_2 : \alpha_2; \alpha'_2) \doteq e}{\exists\, \alpha'.\, (m_2 : \alpha_2; \alpha'_2) \doteq e \wedge \alpha'_1 \doteq (m_2 : \alpha_2; \alpha') \wedge \alpha'_2 \doteq (m_1 : \alpha_1; \alpha')}$$

(1) In Rule Decompose, f is any type symbol, including $(m : \_; \_)$ as well.
(2) To ensure termination, rule Generalize must be restricted to the case where τ is not a variable and α appears in e but not as a term variable of e.

Figure 6.5: Unification as solving multi-sets of multi-equations

$$(\texttt{fun}\ (x)\ a : \tau_1 \to \tau_2 <: \tau'_1 \to \tau'_2) \longrightarrow \langle m_i = (v \# m_i : \tau_i <: \tau'_i)^{i\in J} \rangle \longrightarrow \texttt{fun}\ (x)\ (a[(x : \tau'_1 <: \tau_1)/x] : \tau_2 <: \tau'_2)$$

The counterpart is that types, although not actively participating, would be kept during reduction. The formulation we have chosen has a simpler semantics and makes it clearer that the reduction is actually untyped.

The soundness of the language is stated by the two following theorems.

Theorem 13 (Subject Reduction) *Reduction preserves typings* (i.e. for any A, if $A^\star \vdash a : \tau$ and $a \longrightarrow a'$ then $A^\star \vdash a' : \tau$.)

Theorem 14 (Normal forms) *Well-typed irreducible normal forms are values* (i.e. if $\emptyset \vdash a : \tau$ and a cannot be reduced, then a is a value.)

See the appendix 6.3 for proofs of these theorems.

These results easily extend to cope with constants, as in core ML, provided δ-rules for constants are consistent with their principal types.

6.6 Type inference

Types of Objective ML are a restriction of record types. First-order unification for record types is decidable, and solvable unification problems admit principal solutions, even in the presence of recursion [108].

The unification algorithm is a simplification of the one used in ML-ART [108]. It is described in figure 6.5 as a rewriting process over unification problems. This formalism was introduced in [57] and has already been used for record types in [104]. A unification problem also called a *unificand*, is a multi-set of multi-equations preceded by a list of existentially quantified variables. It is written $\exists\, \alpha_1, \ldots \alpha_p.\, e_1 \wedge \ldots e_q$. A *multi-equation* e is a multi-set of types written $\tau_1 \doteq \ldots \tau_n$. The algorithm assumes that recursive types $\mu\alpha.\tau$ have been encoded using equations $\exists\, \alpha.\, \alpha \doteq \tau$.

A substitution is a solution of a multi-equation if it makes all its types equal. A solution of a unificand is the restriction of a common solution to all its multi-equations outside of the existentially quantified variables.

Unificands can be simplified by applying the rewriting rules given in figure 6.5. Structural rules have been omitted: they include associativity and commutativity of both $\wedge$ and $\doteq$ and the extrusion and renaming of existential variables. Rules FUSE, DECOMPOSE and GENERALIZE are standard. Rule FUSE merges two multi-equations that have a variable in common. Rule DECOMPOSE decomposes terms of a multi-equations into smaller ones. Rule GENERALIZE splits terms into smaller terms. Thus, unificands can always be rewritten so that terms are of depth at most one. This permits maximal sharing during unification. It also ensures termination of rewriting in the presence of recursive types. The only difference with unification in a free algebra is the mutation rule MUTE for left-commutativity. It identifies two terms $(m_1 : \tau_1; \tau_1')$ and $(m_2 : \tau_2; \tau_2')$ with different top symbols $(m_1 : \_; \_)$ and $(m_2 : \_; \_)$ provided their equality can be established by the application of an axiom at the root.

The algorithm proceeds by rewriting multi-sets of multi-equations according to the above rules. Each step preserves the set of solutions. Moreover, the process always terminates, reducing any unificand to a canonical form.

A unificand is in a solved form if all of its multi-equations are merged and each one is fully decomposed (*i.e.* it contains at most one non-variable term). Principal unifiers can be read directly from solved forms. A canonical unificand that is not in a solved formed contains a clash (two incompatible types that should be identified) and is not solvable.

The framework and the meta-theory of unificands are standard. The equational theory of object types is a sub-case of the more general algebra of records types; for details and proofs, the reader is referred to [104].

Objective ML does not allow classes as first-class values. Indeed, in the expression `fun` (x) a, variable x cannot be bound to a class (or a value containing a class). Thus, class types never need to be guessed. Polymorphism is only introduced at LET bindings of classes or values. This ensures that type inference reduces to first-order unification, as it is the case in ML. Consequently, Objective ML has the principal type property. Type inference for classes is straightforward. The links between first-order unification, type inference and principal types are described in a more general setting in [102].

Theorem 15 (Principal types) *For any typing context A and any program a that is typable in the context A, there exists a type τ such that $A \vdash a : \tau$ and for any other type τ' such that $A \vdash a : \tau'$ there exists a substitution θ whose domain does not intersect the free variables of A and such that $\tau' = \theta(\tau)$.*

6.7 Abbreviation enhancements

Object types tend to be very large. Indeed, the type of an object lists all its methods with their types, which can themselves contain other object types. This quickly becomes unmanageable [108, 40]. Introducing abbreviations is thus of crucial importance. This section presents the general abbreviation mechanism of Objective ML and the next section focuses on abbreviating object types. The simple type abbreviation mechanism of ML is not sufficiently powerful: abbreviations are expanded and lost during unification and they do not interact well with recursive types. Several improvements have thus been made to the abbreviation mechanism. First, abbreviations are kept during unification and propagated as much as possible. Second, a larger class of abbreviations are

accepted: abbreviations can be recursive and their arguments can be constrained to be instances of some given types.

In our implementation, types are considered as graphs. In particular, when two types are unified, they become identical rather than two separate, equal types. A construct has been added to the syntax to express type graphs: the construct (τ `as` α) is used to bind α to τ, similarly to the notation `rec` $\alpha.\tau$. However, a main difference is that with aliases α is also bound outside of τ. As an example, the two types $(\langle m : \alpha\rangle$ `as` $\alpha') \to \alpha'$ and $\langle m : \alpha\rangle \to \langle m : \alpha\rangle$ are different graphs, that represent the same regular tree. There are two reasons for considering types as graphs. First, unification rolls types. For instance, unifying types $\tau = \alpha$ and $\tau' = \langle m : \alpha\rangle$ results in type $\tau = \tau' = (\langle m : \alpha\rangle$ `as` $\alpha)$, rather than instantiating α to $\langle m : \alpha'\rangle$ `as` α' in both types (in the later case, τ' would become $\langle m : \langle m : \alpha'\rangle$ `as` $\alpha'\rangle$). Second, unification propagates abbreviations. Abbreviations can be considered as names for nodes. Unifying an abbreviated type with another type makes both types being abbreviated. For instance, unifying the argument of a functional type to an abbreviated type may propagate the abbreviation to the result type. This is demonstrated in the following example.

```
let bump x = x#move 1; x;;
value bump :
  (⟨ move : int → β; .. ⟩ as α) → α =
  ⟨fun⟩
```

Nodes are shared between the argument type and the result type. The ellipsis stands for an anonymous row variable. When typing the expression `bump p` below, type (⟨`move : int` $\to \beta$; `..`⟩ `as` α) and type `point` are identified. The type of `bump p` is thus also abbreviated to `point`.

```
let p = new point 7;;
value p : point = ⟨obj⟩
bump p;;
- : point = ⟨obj⟩
```

Not all the sharing is exposed to the user : sharing reveals too much of useless information. So, only aliasing of open object types (thus row variables can be printed as ellipses) and aliasing defining recursive types are printed. It would be possible to remove some aliasing during type generalization, so that printed types would exactly reflect their internal representations. However, this would complicate the implementation needlessly.

Abbreviations can be recursive. That is, in the definition of the abbreviation `type` $(\bar\alpha)\ \kappa = \tau$, the type constructor κ may occur in the body τ, as long as all occurrences have the same parameters $\bar\alpha$. This restriction is extended to mutually recursive abbreviations. It ensures that abbreviations expand to regular trees. In the implementation, any type constructor standing for an abbreviation caches the expansions of abbreviations it appears in. Thus, when an abbreviation is expanded several times during the traversal of a type, it expands each time to the same type.

Type abbreviations are generalized to allow constraints on the type parameters of the abbreviations. This is an extension to the abbreviations of LCS [7], that were also used in [108]. In an abbreviation definition, parameters are types rather than type variables: `type` $(\bar\tau)\ \kappa = \tau_0$. All free variables of τ must be bound in $\bar\tau$. Actual arguments of an abbreviation must always be instances $\theta(\bar\tau)$ (for some substitution θ) of the parameters $\bar\tau$. Then, the abbreviation can expand to type $\theta(\tau_0)$. For instance, if the type constructor κ is defined as `type` $(\alpha * \alpha')\ \kappa = \alpha \to \alpha'$, then (`int` $*$ `bool`) κ will expand to `int` $\to$ `bool`. To expand an abbreviation, the arguments are usually substituted

for the parameters. Instead, we choose to unify the arguments with the corresponding parameters. The constraints need only to be enforced when parsing a type given by the user. Then, expansion is guaranteed to succeed. Indeed, a substitution θ can always be applied to an abbreviation $(\bar{\tau})\ \kappa$. The expansion of $\theta((\bar{\tau})\ \kappa)$ is equal to the result of applying the substitution θ to the expansion of $(\bar{\tau})\ \kappa$. In particular, constraints are preserved by substitution.

6.8 Abbreviating object types

We will now describe how the abbreviation mechanism presented in the previous section is used to generate abbreviations for objects. This mechanism is used to automatically abbreviate object constructors: the expression `new` z will have type $\tau_1 \to \ldots \to \tau_n \to (\tau'_i)\ \kappa_z$, where κ_z is the abbreviation associated with class z.

General type abbreviations, introduced in the previous section, can be used to simplify object types. Rather than sorting types to ensure that object types are well-formed, we require the stronger condition that any two object types that share the same row variable must be equal. This eliminates incorrect types such as $\langle\rho\rangle \to \langle m : \tau; \rho\rangle$. Types such as $\langle m : \tau_1; \rho\rangle \to \langle m : \tau_2; \rho\rangle$, at the basis of record extension, are also rejected. However, no primitive operation on objects can exhibit such a type. These types can thus be ruled out without seriously restricting the language. Moreover, all programs keep the same principal types. This restriction was implemented to avoid explaining sorts to the user. It also makes the syntax for types somewhat clearer, as row variables can then always be replaced by ellipsis. Furthermore, sharing can still be described with aliasing. For instance, $\langle m : \tau; \rho\rangle \to \langle m : \tau; \rho\rangle$ is written $(\langle m : \tau; ..\rangle$ `as` $\alpha) \to \alpha$.

A class definition `class` $z = c$ `in` ... automatically generates an abbreviation for the type of its instances. For specifying it, one actually needs to add type parameters to the class definitions, corresponding to the one of the abbreviation. That is, we should write

$$\texttt{class}\ (\bar{\alpha})\ z = c\ \texttt{in}\ \ldots \tag{1}$$

where the parameters $\bar{\alpha}$ must appear in c.

In fact, abbreviations are generated from class *types*. It follows from type inference that the class definition c has a principal class type $\tau'_0 \to \ldots \to \tau'_n \to \texttt{sig}\ (\tau_y)\ \varphi\ \texttt{end}$. Here, τ_y is the type matched by objects in all subclasses. It is always of the form $\langle m_i : {\tau_i}^{i\in I}; \tau\rangle$ where `method` (φ) is a subsequence of $m_i : {\tau_i}^{i\in I}$ and τ is either $\emptyset$ (this is a pathological case, where the class cannot be extended with new methods) or a row variable ρ. If `method` (φ) is exactly $m_i : \tau_i$, then it is possible to create objects of that class; they will have type $\tau_y[\emptyset/\rho]$. Otherwise, the class is virtual and can only be inherited in other class definitions. If all free type variables of τ_y except ρ are listed in $\bar{\alpha}$, we automatically define two abbreviations:

$$\texttt{type}\ (\bar{\alpha}, \rho)\ \#\kappa_z = \tau_y \qquad\qquad \texttt{type}\ (\bar{\alpha})\ \kappa_z = (\bar{\alpha}, \emptyset)\ \#\kappa_z$$

The former matches all objects of subclasses of c. The latter is a special case of the former, and abbreviates any objects of class c.

Let us consider an example. Class `point` has type `int` $\to$ `sig` $(\langle$`move` : `int` $\to$ `int`$; \rho\rangle)\ \varphi$ `end` for some φ whose only method is `move` : `int` $\to$ `int`. Thus, class point is not virtual. The two following abbreviations are generated for this class:

$$\texttt{type}\ \rho\ \#\texttt{point} = \langle\texttt{move} : \texttt{int} \to \texttt{int}; \rho\rangle \qquad\qquad \texttt{type}\ \texttt{point} = \langle\texttt{move} : \texttt{int} \to \texttt{int}\rangle$$

One can check that the type `point` is indeed an abbreviation for the type of objects of the class `point`, and that the type of an object of any subclass of the class `point` is an instance of the type ρ #`point`.

In the concrete syntax, the row variable ρ is treated anonymously (as an ellipsis) and is omitted. The former abbreviation $\#\kappa_z$ is given a lower priority than the regular ones in case of a clash. It also vanishes as soon as the row variable is instantiated, so as to reveal the value taken by the row variable.

In fact, we allow κ_z and $\#\kappa_z$ to occur in the definition of b. The previous definitions can be rewritten to handle the general case correctly.

Constrained abbreviations are natural for abbreviating objects, as, for instance, a sorted list of comparable objects should be parameterized by the type of its elements, which in turn is not a type variable. Moreover this extension makes it possible to avoid row variables as type parameters (as the whole object type can appear as a parameter).

Constrained type abbreviations are also convenient since, in a class definition `class` $(\bar{\alpha})\ z = c$ `in` ..., class type parameters $\bar{\alpha}$ may have been instantiated to some types $\bar{\tau}_\alpha$ while inferring the class type $\tau'_0 \to \ldots \to \tau'_n \to$ `sig` $(\tau_y)\ \varphi$ `end`. The two abbreviations generated by the class definition are thus:

$$\texttt{type}\ (\overline{\tau_\alpha}, \rho)\ \#\kappa_z = \tau_y \qquad\qquad \texttt{type}\ (\bar{\alpha})\ \kappa_z = (\bar{\alpha}, \emptyset)\ \#\kappa_z$$

The latter is unchanged except that the constraints of the first ones are implicit in the second one.

Class types are shown to the user stripped of their type parameters. The parameters that constraint the type abbreviations are described by constraint clauses:

```
class α circle (p : α) = struct
  field point = p
  method center = point
  method move m =
    if m = 0 then 0 else
    point#move (1 + Random.int m)
end;;
class α circle : α → sig
  constraint α = ⟨ move : int → int; .. ⟩
  field point : α
  method center : α
  method move : int → int
end
```

This class defines the abbreviation

$$\texttt{type}\ (\langle \texttt{move} : \texttt{int} \to \texttt{int}; \rho \rangle\ \texttt{as}\ \alpha)\ \texttt{circle} = \langle \texttt{center} : \alpha; \texttt{move} : \texttt{int} \to \texttt{int} \rangle$$

As a result of the abbreviation mechanisms, type inference may reject some class definitions whose principal types have free variables. For instance, the following variant of class `point` is rejected, since the method `getx` is polymorphic and therefore the class should be parametric.

```
class point x0 = struct
  field x = x0
  method getx = x
end;;
```

Of course, one could choose an arbitrary ground class type, for instance:

```
class point : int → sig
  field x : int
  method getx : int
end
```

Any other ground type could be used instead of `int`. We decide to reject those programs. This preserves the property that any typable program has a principal type —and all other useful properties of the type system.

This phenomenon is not new. It already appeared in several extensions of ML. Imperative constructs limit polymorphism. Thus, some variables that are not generalizable may occur in the type of a toplevel expression. In such a case, most languages would reject the program. For instance, the extension to ML with dynamics [72] rejects `fun x → dynamics x`, since the dynamic type of `x` in `dynamics x` is statically unknown.

All the examples above would have principal types as long as type inference is concerned. We can argue that some programs have been rejected for sake of simplicity and uniformity of the language, but not because of a failure of type inference: For instance, in Objective ML we could just omit the corresponding abbreviation whenever some type parameter is missing, and print a warning message instead of an error message.

6.9 Extensions

This section lists other useful features of Objective ML that have been added to the implementation. Imperative features have been ignored in the formal presentation since their addition is theoretically well-understood and independent of the presence of objects and classes. Other features are less important in theory, but still very useful in practice: private instance variables, coercion primitives.

Before we explore these extensions, let us consider an interesting restriction of the language. If recursive types are only allowed when the recursion traverses an object type, Objective ML becomes a conservative extension of ML, which we claimed in the introduction. Of course, all ML programs can be defined, and behave similarly. Moreover, programs that are syntactically ML programs are now well-typed ML programs if and only if they are well-typed in Objective ML. However, in the implementation Objective Caml, the presence of modules requires the use of recursive abstract types as well. This is because recursive object types may be abstracted. Thus, Objective Caml is not strictly speaking a conservative extension of ML. Still, it is a conservative extension of ML with recursive types.

6.9.1 Imperative features

We have intentionally used references in the very first example. We did not formalize references in the presentation of Objective ML, since we preferred to keep the presentation simple and put emphasis on objects and classes. The addition of imperative features to Objective ML is theoretically as simple and as useful practically as their addition to ML. Both the semantics and the properties of reduction with respect to typing extend to operations on the store without any problem. The formalization copies the one for core ML.

In fact, the implementation Objective Caml also allows fields to be mutable in a similar way mutable record fields are treated in Caml [73]. For instance, we could have written:

```
class point x0 = struct
```

```
  field mutable x =  x0
  method move d = (x ← x + d; x)
end;;
class point : int → sig
  field mutable x : int
  method move : int → int
end
```

Objective Caml only allows generalization of values (actually, a slightly more general class of non expansive expressions). The creation of an object from a class `c` is not considered as a value (as it is the application of function `new c` to some arguments). Mutable fields in classes are typed as any other fields, except that mutability properties are also checked during typechecking.

6.9.2 Local bindings

As shown by the evaluation rules for objects, both value and method components are bound to their rightmost definitions. All value components must still be evaluated even though they are to be discarded.

Object-oriented languages often offer more security through private instance variables. The scope of a field can be restricted so that the field is no more visible in subclasses.

This section presents local bindings, that are only visible in the body of the class they appear in. This is weaker than what one usually expects from private fields, as a class cannot, for instance, inherit a field and hide it from its subclasses (see section 6.10.1).

The syntax is extended as follows:

$$\begin{aligned} d &::= \ldots \mid \texttt{local}\ x = a\ \texttt{in}\ b \\ F_d &::= \ldots \mid \texttt{local}\ x = E\ \texttt{in}\ b \end{aligned}$$

with the corresponding typing rule:

$$\frac{A^\star \vdash a : \tau \qquad A + x : \tau \vdash b : \varphi}{A \vdash \texttt{local}\ x = a\ \texttt{in}\ b : \varphi}\ (\textsc{Local})$$

Local bindings are reduced top-down, like inheritance:

$$\texttt{local}\ x = v\ \texttt{in}\ b; b' \longrightarrow b[v/x] + b'$$

In practice, however, local bindings would rather be compiled as anonymous fields. This would make methods independent of local bindings.

Initialization parameters could also be seen as local bindings in the whole class body, and could also be compiled as anonymous instance variables. For instance, the definition

```
class point y = struct method x = y end;;
```

could be automatically transformed into the equivalent program:

```
class point y = struct
  local y = y in method x = y
end;;
```

That way, the method `x` becomes independent of the initialization parameter `y`. Then, classes can be reduced to class values: inheritance is reduced to local bindings, local bindings are flattened, and method overriding is resolved.

6.9.3 Coercion primitives

Explicit coercions require both the domain and co-domain to be specified. This eliminates the need for subtype inference. In practice, however, it is often sufficient to indicate the co-domain of the coercion only, the domain of the coercion being a function S of its co-domain.

For convenience, we introduce a collection of coercion primitives:

$$(\_ <: \tau) : \forall\, \bar{\alpha}.\, S(\tau) \to \tau$$

where $\bar{\alpha}$ are free variables of $S(\tau)$ and τ, and $S(\tau)$ is defined as follows:

- We call positive the occurrences of a term that can be reached without traversing an arrow from the left hand side. (This is more restrictive than the usual definition, where the arrow is treated contravariantly).
- For non recursive terms, we define $S_0(\tau)$ to be τ where every closed object type that occurs positively is opened by adding a fresh row variable.
- Terms with aliases are viewed as graphs, or equivalently as pairs of a term τ_0 and a list of constraints $\alpha_i = \tau_i$.

  Let θ be a renaming of variables α_i into fresh variables.

  Let τ_i' be τ_i in which every positive occurrence of each α_i is replaced by $\theta(\alpha_i)$.

  We return $(S_0(\tau_0'), \{\theta(\alpha_i) = S_0(\tau_i'), i \in I\} \cup \{\alpha_i = \tau_i, i \in I\})$ for $S(\tau)$.

For example,

$$S(\langle m_1 : \langle m_2 : \mathtt{int}\rangle \to \langle m_3 : \mathtt{bool}\rangle\rangle) = \langle m_1 : \langle m_2 : \mathtt{int}\rangle \to \langle m_3 : \mathtt{bool}; \rho_3\rangle; \rho_1\rangle$$

$$S(\langle m : \alpha\rangle \ \mathtt{as}\ \alpha) = \langle m : \alpha'; \rho\rangle \ \mathtt{as}\ \alpha'$$

$$S(\langle m : \alpha \to \alpha\rangle \ \mathtt{as}\ \alpha) = \langle m : (\langle m : \alpha \to \alpha\rangle \ \mathtt{as}\ \alpha) \to \alpha'; \rho\rangle \ \mathtt{as}\ \alpha'$$

The operator S has the two following properties:

$$(1) \quad S(\tau) \le \tau \qquad\qquad (2) \quad \exists\theta\ (\theta(S(\tau)) = \tau \wedge \theta(\tau) = \tau)$$

The former gives the correctness of the reduction step $(a <: \tau) \longrightarrow (a : S(\tau) <: \tau)$. The later shows that if a has type τ then $(a <: \tau)$ also has type τ.

There is no principal solution for an operator S satisfying (1). Consider τ to be $\langle m : \mathtt{int}\rangle \to \mathtt{int}$. There are only two solutions, $\langle m : \mathtt{int}\rangle \to \mathtt{int}$ and $\langle\rangle \to \mathtt{int}$ and none is an instance of the other. This counter-example shows the weakness of the simulation of subtyping with row variables, especially on negative occurrences. There are other examples of failure on positive occurrences, but only using recursive types. For instance, if τ is $\langle x : \alpha\rangle \ \mathtt{as}\ \alpha$, then both $\langle x : \tau; \rho\rangle$ and $\langle x : \beta; \rho'\rangle \ \mathtt{as}\ \beta$ are solutions for $S(\tau)$, but no solution is more general than both of these. Our choice of S (and correspondingly, our choice of coercion primitives) is somehow arbitrary, but works well in practice. This justifies the exclusion of semi-explicit coercions from the core language, but leave them as a collection of primitives. In fact, most coercions are of the form $(a : S(\tau) <: \tau)$. Thus, the domain of a coercion rarely needs to be given.

6.10 Future work

This short section describes three possible extensions of importance to Objective ML. Each extension requires further theoretical and design investigation before it can be integrated within the language Objective Caml.

6.10.1 Restriction of class interfaces

In section 6.9.2 we have shown that field components can be declared local to a class. However, this does not enable class components to be hidden *a posteriori*. Assume, for instance, that a library provides an implementation of a class z with two fields x and x' and two methods m and m'. A module may define a class z'' that inherits from an imported class z' whose interface is a restriction of the one of the class z to the field x and the method m only. Can class z be used as an import to the module? This problem corresponds to a common situation of interface restriction when reusing code. However, interface restriction is not currently possible.

Private fields would actually not be difficult to hide. However, hiding methods in subclasses conflicts with late binding and a flat method name space. For instance, assume, method m' is implicitly hidden when inherited in class z'', and that class z'' defines a method m', possibly with another type!

Clearly, when a method m is hidden in a class z, self-invocations of m in all other methods of z should be replaced by calls to a function representing the method m. This is a complex operation that is difficult to compile.

Another problem is that method m' appears in the type of `self`. Hiding the method thus requires to modify *a posteriori* the type of `self`. This would not be correct if, for instance, this type is the type of a method argument.

A partial solution is to give each method a different view of `self` inside classes. This is usually the case when classes are treated as a collection of pre-methods. Another choice, weaker but still useful, is to split the input and output view of `self`. The former lists the methods that are required while the later enumerates methods that are provided. However, in the presence of type inference, such solutions tend to increase the size of a class to a point that may become unreadable [108]. The gain in expressiveness is also weakened by a later detection of errors. Clearly, it is an error if a method has incompatible required and provided types. However, this would only be detected when the object is created. In the design of Objective ML, we have deliberately limited the expressiveness of class types to keep them readable. Many variations are theoretically possible, but very few of them seem to improve expressiveness significantly without sacrificing simplicity.

Another possibility is to introduce private methods. They would not appear in the type of `self`, consequently, they should be invoked differently. Private methods could have the same scope as fields. In particular, they could be hidden *a posteriori* as well.

The addition of *final* classes could also resolve the problem. These classes could not be inherited. Then, a class could be soundly matched against a final class interface that omits some of its methods.

6.10.2 Polymorphic methods

In a classical programming style, functions and data are clearly separated. Functions are often polymorphic and thus can be applied uniformly to different kinds of data. Data may be structured. It very rarely carries functions, and is usually monomorphic. In objects, data and methods are jointly defined and stored or passed as arguments together — at least from a theoretical point of view.

Let-bound toplevel functions often become methods of λ-bound first-class objects. Unfortunately, polymorphism is lost during this transformation. For instance, a class implementing sets, would naturally provide a fold method. The inferred class type would be of the form:

```
class α set = struct  ...
  method fold : (α → β → β) → β → β
end
```

However, this is rejected, since variable β is unbound in α `set`. An attempt to fix the problem would be to parameterize the class `set` over β as well, that is, to replace α `set` in the definition above by (α,β) `set`. However, this is not very intuitive, since the object stays parametric in β even when all its fields have a ground type. Moreover, the method `fold` becomes monomorphic and thus can only be applied to functions of the same type, whenever the object is λ-bound.

The intuition is of course that the method `fold` should be polymorphic. That is, the class `set` should have the following class type:

```
class α set = struct ...
  method fold : All β. (α → β → β) → β → β
end
```

The addition of polymorphic methods could also be used to reduce the number of explicit coercions. In a class definition methods may have types more polymorphic than expected. For instance, assume that class point has type:

```
class point (int) = struct
   field x : int  method getx : int
end;;
```

Then, the following subclass of `point` will not typecheck:

```
class eq_point x = struct
  inherit point x
  method eq p = p#getx = self#getx
end;;
```

The parameter `p` of the method `eq` does not need to be a point but an object with method `getx` of type `int`. Thus, its type $\langle$`getx : int; ..`$\rangle \to$ `bool` has a free row variable. As for the case of `set`, the row variable in the type of `p` can be bound in in a constraint type parameter as follows:

```
class α eq_point x = struct
 inherit point x
 method eq (p:α) = p#getx = self#getx
end;;
class α eq_point : int → sig
  constraint α = ⟨ getx : int; .. ⟩
  field x : int
  method getx : int
  method eq : α → bool
end
```

Again, this is not very intuitive and one might prefer to add a stronger type constraint. One choice is to require `p` to be of the same type as `self`. However, this unnecessarily makes `eq` a binary

method and so restricts its further use with arguments of type `eq_point` only. Constraining `p` to be a `point` in the definition of the method `eq` is another possibility:

```
class eq_point x = struct
 inherit point x
 method eq (p:point) = p#getx = self#getx
end;;
class eq_point : int → sig
  field x : int
  method getx : int
  method eq : point → bool
end
```

This solution is more general, although it usually requires explicit coercion when invoking the method `eq`:

```
let p = eq_point 1 in p#eq (p ⟨: point);;
```

Polymorphic methods would allow a more natural class type for the `eq_point` (first definition):

```
class eq_point : int → sig
  field x : int
  method getx : int
  method eq p :
      All (⟨getx : int; ..⟩ as α). α → bool
end;;
```

Moreover, thanks to the polymorphic (anonymous) row variable, messages could then be sent to the method `eq` with an argument of type either `point` or `eq_point`.

We consider that the lack of polymorphic methods is a weakness of Objective ML. We believe that polymorphic methods would make most explicit coercions unnecessary.

Some solutions to extend ML with first class-polymorphism already exist in the literature. Simple but rudimentary proposals can be found in [108, 83] and better integration of first-class polymorphism inside Objective ML has recently been studied in [46].

6.10.3 Integrating classes and modules

Objects and classes of Objective ML are orthogonal to the other extensions of ML. In particular, the module system of ML extends directly to classes and objects [69]. Indeed, the implementation of Objective ML, called Objective Caml [70], offers a rich language of both modules and classes. Classes and modules share a lot of properties: they offer some form of abstraction; they also help structuring large applications; and they facilitate reusability of code. In fact, they are quite different. Modules are a very general and powerful abstraction. However, it is difficult to allow recursion between several modules or to give a meaning to self inside modules. On the other hand, classes are a much more specialized paradigm that has proved extremely convenient for some applications. Objects find their limitation with multiple dispatch. Hiding components also remains a difficult task.

For historical reasons, libraries of Objective Caml are implemented as modules. In practice, many of these libraries could be rewritten as classes. Choosing one style or another is not insignificant, since it is a global commitment to the architecture of the application. The class version and the module version of the same libraries are very similar, but their code cannot currently be

shared. This is, of course, unsatisfactory. We hope that more work will allow a better integration of modules and classes.

6.11 Comparison to other works

The work closest to Objective ML is ML-ART [108]. Here, object types are also based on record types and have similar expressiveness. State abstraction is based on explicit existential types in ML-ART; in Objective ML, it is obtained by scope hiding, but it could also be explained with a simple form of type abstraction. No coercion at all is permitted in ML-ART between objects with different interfaces. Unfortunately, ML-ART has no type-abbreviation mechanism. This was a major drawback, which motivated the design of Objective ML. On the other hand, classes are first class values in ML-ART. We, however, do not think this is a major advantage. The restriction is a deliberate choice in the design of Objective ML, to keep the language simpler. In theory, most features of ML-ART could have been kept in Objective ML. In practice, however, it would have changed the language significantly.

Another simplification in Objective ML is that in classes all methods view self with the same type. This is not required by the semantics, and could technically be relaxed by making method types more detailed in classes (see [108]). We found that this extra flexibility is not worth the complication of class types.

Our object types are a simplification of those used in [106]. The simplification is possible since object types are similar to record types for polymorphic access, and do not require the counterpart of record extension. Moreover, as discussed above, our implementation assumes the stronger condition that two object types sharing the same row variable are always identical. With this restriction, object types seem to be equivalent to kinded record types introduced in [84]. Ohori also proposed an efficient compilation of polymorphic records (which does not scale up to extensible records) in [85]. However, his approach, based on the correspondence between types and domains of records cannot be applied to the compilation of objects with code-free coercions.

Objects have been widely studied in languages with higher-order types [19, 82, 16, 2, 93, 14]. These proposals significantly differ from Objective ML. Types are not inferred but explicitly given by the user. Type abbreviations are also the user's responsibility. On the contrary, all these proposals allow for implicit subtyping.

Our calculus differs significantly from Abadi's and Cardelli's primitive calculus of objects mostly as a result of design choices. We have chosen primitive classes because inferred types of sets of pre-methods would be complex to be readable (see [108] for instance). We have emphasized the role of row variables because we have chosen not to infer subtyping, therefore avoiding the more complicated framework of constraint types. On the other hand we have included other features such as instance variables, to avoid their encoding as methods not involving self, and to keep with the more simple state-abstraction mechanism by scope hiding. Technically a major difference, Objective ML does not allow method overriding.

Open record types are connected to the notion of matching introduced by Kim Bruce [16, 18]. Matching seems to be at least as important as subtyping in object-oriented languages. Row variables in object types express matching in a very natural way. While explicit matching may require too much type information, type inference makes object matching very practical.

Palsberg has proposed type inference [87] for a first-order version of Abadi and Cardelli's calculus of primitive objects [1]. However, that language is missing important features from the higher-order version [2]. Type inference is based on subtyping constraints and the technique is similar to the one used in [40]. This later proposal [40, 41] is closer to a real programming language, and more suited

for comparison. Here, the authors use a subtyping relation that is more expressive than ours, as they can prove subtyping under some assumptions. They can also infer coercions. However, the types they infer tend to be too large. Indeed, they do not have an abbreviation mechanism. Their inheritance is weaker than ours since they must explicitly list all inherited methods in subclasses. We think the two proposals are complementary and could benefit from one another. In particular, it would be interesting to adapt automatic type abbreviations to constraint types. The problem is still non-trivial since inferred type-constraints are hard to read even in the absence of objects.

The remainder of this section is dedicated to the comparison with three other proposals for adding objects to ML. They all use implicit subtyping, which is, however, restricted to atomic structural subtyping [79, 44]. As a result, they all have the same difficulty with parameterized classes, making it impossible to relate objects created from classes with a different number of parameters, even when the objects have the same interface. For instance, objects of a class `string` are of incompatible type with objects of a parameterized class `vector` when the parameter type is character.

In [14], Bourdoncle and Metz propose a language based on some restricted form of type constraints [41]. However, they do not provide type inference.

The two following proposals include type inference; however, fully polymorphic method invocation cannot be typed. Two different solutions are proposed; they both amount to providing some explicit type information at method invocation.

More precisely, in Duggan's proposal [38], methods must be predeclared with a particular type scheme. Thus methods carry type information alike data-type constructors in ML. For instance, `move` would be assigned type scheme $\forall\, \alpha_y.\, \alpha_y \to int$. Type schemes that are assigned to methods are polymorphic in α_y: they are arrow types whose domain is always a variable α_y, standing for the type of self. Object types only list the methods that objects of that type must accept. For instance, `point` would be given type $\langle \texttt{move} \rangle$. The user must provide more type information that in Objective ML. The same method name cannot be used in two different objects with unrelated types. Objects of parameterized classes are treated especially, using constructor kinds. As mentioned above, objects of a parameterized class reveal forever that they are parameterized. For instance, let us consider a class of vectors parameterized over the type α. All methods of that class must be given a type scheme of the form: $\forall\, {\alpha_\kappa}^{Type \to Type}.\, \forall\, \alpha.\, \alpha\ \alpha_\kappa \to \tau$, where variable α_κ range over type constructors. That is, instead of the type τ_y of `self`, only the type constructor κ of the type τ_y is hidden. This reveals the dependence of τ_y on its parameters, and the parameters themselves. Methods of parameterized classes are incompatible with methods of non-parameterized classes. Objects of a vector class of characters cannot be related to objects of a string class even though they might have the same interface. In Objective ML, two such objects could be mixed. However, Objective ML does not allow polymorphic methods while Duggan's proposal does. A polymorphic method `map` could be declared with type scheme: $\forall\, {\alpha_\kappa}^{Type \to Type}.\, \forall\, \alpha.\, \forall\, \alpha_1.\, \alpha\ \alpha_\kappa \to (\alpha \to \alpha_1) \to \alpha_1\ \alpha_\kappa$. Intuitively, `map` carries implicit universal intros and elims, like data constructors carry arguments of existentially or universally quantified types in [64, 108, 83]. Recursive kinds actually allow some form of polymorphism that is different from polymorphic methods discussed in section 6.10.

In Object ML [115], Reppy and Riecke treat objects as a generalized form of concrete data-types. Types are also inferred in Object ML, but the authors do not claim a principal type property. Also, method invocation must always mention the class of the object to which the method belongs. Each object is actually tagged with a constructor that carries the class the object originated from. Therefore, objects can be tested for membership to some arbitrary class in some inheritance relationship. Only single inheritance is allowed. The subtyping relationship between objects is declared and corresponds to the inheritance forest. Classes are generative, that is, objects

of different classes have different types. Although these types can be related by subtyping, they are never in an instance relationship. Some object coercions, but apparently not all, are implicit. On the contrary in Objective ML, classes are transparent, that is, objects types are structural and only describe the interface of objects: two objects with exactly the same interface have equal types. Two objects of classes in a subclass relationship are not necessarily related, but when they are, one type is simply an instance of the other. Object ML does not provide any inheritance mechanism, except by means of encodings [114]. Typing of binary methods is also a problem, which is solved via runtime class-type tests.

Conclusion

Objective ML has been designed to be the core of a real programming language. Indeed, the constructs presented here have been implemented in the language Objective Caml. We chose class-based objects since this approach is now well understood in a type framework and it does not require higher-order types.

The original part of the design is automatic abbreviation of object types. Although this is not difficult, it is essential for making the language practical. It has been demonstrated before that fully inferred object types are unreadable [108, 40]. On the contrary, types of Objective ML are clear and still require extremely little type information from the user. To our knowledge, all other existing approaches require more type declarations.

Objective ML is also interesting theoretically for the use of row variables [119, 106]. Row variables are very close to matching and seem more helpful than subtyping for the most common operations on objects. Message passing and inheritance are entirely based on row variables, which relegates subtyping to a lower level.

Another interesting aspect of our proposal is its simplicity. This is certainly due to the fact that Objective ML is very close to ML. Specifically, most features rely only on ML polymorphism. This leads to very simple typing rules for objects and inheritance. Coercions, based on subtyping, can be explained later. Data abstraction is guaranteed by scope hiding rather than by type abstraction; this is a less powerful but simpler concept.

The main drawback of Objective ML is the need for explicit coercions. Coercions are necessary. However, we think they occur in few places. Thus, explicit coercions should not be a burden. Furthermore, coercions could in theory be made implicit using constraint-based type inference.

In our implementation of Objective ML, classes and modules are fully compatible, but orthogonal. That should be particularly interesting to compare these two styles of large-scale programming, and help us to better integrate them. This is an important direction for future work.

Acknowledgment

We thank Rowan Davies who collaborated in the implementation and the design of a precursor prototype of Objective ML.

Appendix

6.1 Typing rules for core ML

$$\frac{x : \forall \bar{\alpha}.\, \tau \in A}{A \vdash x : \tau[\bar{\tau}/\bar{\alpha}]}\ (\textsc{Inst}) \qquad \frac{A + x : \tau \vdash a : \tau'}{A \vdash \texttt{fun}\ (x)\ a : \tau \to \tau'}\ (\textsc{Fun}) \qquad \frac{A \vdash a : \tau' \to \tau \qquad A \vdash a' : \tau'}{A \vdash a\ a' : \tau}\ (\textsc{App})$$

$$\frac{A \vdash a' : \tau' \qquad A + x : \texttt{Gen}(\tau', A) \vdash a : \tau}{A \vdash \texttt{let}\ x = a'\ \texttt{in}\ a : \tau}\ (\textsc{Let})$$

Generalization $\texttt{Gen}(\tau, A)$ is $\forall\, \bar{\alpha}.\, \tau$ where $\bar{\alpha}$ are all variables of τ that are not free in A.

6.2 Binary methods

In Objective ML, it is possible to define binary methods, that is, methods that receive as a parameter an object of the same type as self. Furthermore, a class that has binary methods can be freely extended by inheritance. Of course, binary methods remains binary in a subclass.

The virtual class `comparable` is a template for classes with a binary method `leq`. The component `virtual leq` is a type constraint on the type of self. This method must be applied to an object of the same type as self.

```
class comparable () = struct virtual (α)
  virtual leq : α → bool
end;;
class comparable : unit → sig virtual (α)
  virtual leq : α → bool
end
```

Class `int_comparable` inherits from class `comparable`. It implements method `leq` and adds a method `getx`.

```
class int_comparable (x : int) = struct
  inherit comparable ()
  field x = ref x
  method getx = !x
  method leq o = !x ≤ o#getx
end;;
class int_comparable : int → sig (α)
  field x : int ref
  method leq : α → bool
  method getx : int
end
```

Method `leq` still expects to be applied to an object of the same type as self. So, type $\texttt{int\_comparable} = \texttt{rec}\ \alpha.\langle \texttt{leq} : \alpha \to \texttt{bool}; \texttt{getx} : \texttt{int}\rangle$ is not a subtype of type $\texttt{comparable} = \texttt{rec}\ \alpha.\langle \texttt{leq} : \alpha \to \texttt{bool}\rangle$: inheritance is not subtyping. Indeed, a method `leq` of an object of the

former type expects to be applied to an object that has a method `getx`; this is not ensured by the later type. However, `int_comparable` is an instance of ρ `#comparable`, which is by definition `rec` $\alpha.\langle$`leq` $: \alpha \to$ `bool`$; \rho\rangle$. Binary methods are correctly handled since the type of self is kept open while typing classes: adding the method `getx` to class `comparable` simply amounts to instantiating the row variable in the type of self, to `(getx : int; ..)`. Thus, the type of self in the subclass has a method `getx` and is still open.

As a test, the function `min` will return the minimum of any two objects whose type is an instance of type `#comparable`.

```
let min (x : #comparable) y =
  if x#leq y then x else y;;
value min : (#comparable as α) → α → α =
  ⟨fun⟩
```

This function can thus be applied to objects of type `int_comparable`.

```
let p = min (new int_comparable  7)
            (new int_comparable 11)
in (p, p#getx);;
- : int_comparable * int = ⟨obj⟩, 7
```

6.3 Proofs of type soundness theorems

Subject reduction is a straightforward combination of redex contraction (lemma 30) and context replacement (lemma 25).

Since we have multiple syntactic categories for expressions, contexts, and types, it is convenient to introduce the following meta-notations:

$$\check{a} ::= a \mid b \mid c \mid d \qquad \check{E} ::= E \mid F \mid E_c \mid F_d \qquad \check{\tau} ::= \tau \mid \varphi \mid \gamma$$

These meta-letters are used consistently. For instance, when writing $A \vdash \check{a} : \check{\tau}$, $(\check{a}, \check{\tau})$ means (a, τ), (b, φ), etc, but not (b, τ).

The following propositions are used several times in the proof.

Proposition 21 (Stability by substitution) *If $A \vdash \check{a} : \check{\tau}$, then for any substitution θ, $\theta(A) \vdash \check{a} : \theta(\check{\tau})$.*

Proposition 22 (Extension of environment) *If type environments A and B are identical on free variables of expression a and $A \vdash \check{a} : \check{\tau}$, then $B \vdash \check{a} : \check{\tau}$. If type environment B extends type environment A (that is $B \restriction dom(A)$ is A) and $A \vdash \check{a} : \check{\tau}$, then $B \vdash \check{a} : \check{\tau}$.*

We say that σ is an instance of σ' if any instance of σ is an instance of σ'. We say that type environment A is an instance of type environment A' if both type environments have the same domain and for any element h of their domain $A(h)$ is an instance of $A'(h)$.

Proposition 23 (Strengthening of context) *If type environment A is an instance of type environment B and $A \vdash a : \tau$, then $B \vdash a : \tau$.*

The following lemma somewhat simplifies the proofs.

Lemma 24 (Derivation simplification) *When proving that for all τ, $A_0 \vdash a_0 : \tau$ implies $A \vdash a : \tau$ (for some A_0, a_0, A and a), one can restrict oneself to the case where a derivation of $A_0 \vdash a_0 : \tau$ does not end with rule* SUB. *The general case follows.*

Proof: This is done by induction on the size of derivations. Let us assume that a derivation of $A_0 \vdash a_0 : \tau$ ends as

$$\frac{A_0 \vdash a_0 : \tau' \qquad \tau' \leq \tau}{A_0 \vdash a_0 : \tau} \text{ (SUB)}$$

By induction hypothesis, $A \vdash a : \tau'$. Hence

$$\frac{A \vdash a : \tau' \qquad \tau' \leq \tau}{A \vdash a : \tau} \text{ (SUB)}$$

■

We write $a_1 \subset a_2$ if for any environment A such that $A^\star = A$ and any type τ such that $A \vdash a_1 : \tau$, $A \vdash a_2 : \tau$. Likewise, we write $b_1 \subset b_2$ (resp. $c_1 \subset c_2$) if for any environments A and any class body type φ such that $A \vdash b_1 : \varphi$ (resp. any class type γ such that $A \vdash c_1 : \gamma$), then $A \vdash b_2 : \varphi$ (resp. $A \vdash c_2 : \gamma$). Subject reduction theorem can be restated as follows: if $a_1 \longrightarrow a_2$, then $a_1 \subset a_2$.

Lemma 25 (Context replacement) *For any context E, if $\check{a}_1 \subset \check{a}_2$ then $E[\check{a}_1] \subset E[\check{a}_2]$.*

Proof: The property can be proved independently for each arbitrary one-node context $\check{E}$. Then, the lemma follows by a trivial induction on the size of the context.

Let $\check{E}$ be a one-node context. Let A be a type environment and $\check{\tau}$ a type such that $A \vdash \check{E}[\check{a}_1] : \check{\tau}$ (6.1). We show that $A \vdash \check{E}[\check{a}_2] : \check{\tau}$. Using lemma 24, one can assume that a derivation of (6.1) does not end with rule SUB.

All cases are simple and similar. We show one case for example:

Cas E is `let` $x = []$ `in` a**:** A derivation of (6.1) ends as:

$$\frac{A \vdash a_1 : \tau' \qquad A + x : \mathtt{Gen}(\tau', A) \vdash a : \tau}{A \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a : \tau} \text{ (LET)}$$

By induction hypothesis applied to the first premise, $A \vdash a_2 : \tau'$. Hence $A \vdash \mathtt{let}\ x = a_2\ \mathtt{in}\ a : \tau$

■

The following lemmas (26 thru 29) are used to simplify the proof of redex contraction.

Lemma 26 (Append) *Let A be a typing environment containing no* `super` *bindings. If $A \vdash b_1 : \varphi_1$, $A + (\varphi_1 \setminus \mathtt{method}) \vdash b_2 : \varphi_2$, and φ_1 and φ_2 are compatible (that is, $\varphi_1 \oplus \varphi_2$ is correct), then $A \vdash b_1 @ b_2 : \varphi_1 \oplus \varphi_2$.*

Proof: We actually prove a more general property. Let φ_0 be a sequence of `super` bindings. If $A + \varphi_0 \vdash b_1 : \varphi_1$, $A + (\varphi_1 \setminus \mathtt{method}) \vdash b_2 : \varphi_2$, and φ_1 and φ_2 are compatible (that is, $\varphi_1 \oplus \varphi_2$ is correct), then $A + \varphi_0 \vdash b_1 @ b_2 : \varphi_1 \oplus \varphi_2$.

This is easily proved by induction on b_1. ■

Lemma 27 (Term replacement (variables)) *Let A be a type environment, $\check{a}$ and a' be term expressions, $\check{\tau}$ and τ' be type expressions. If $A^\star \vdash a' : \tau'$ (6.2) and $A + x : \mathtt{Gen}(\tau', A) \vdash \check{a} : \check{\tau}$ (6.3) and bound variables of $\check{a}$ are not free in a', then $A \vdash \check{a}[a'/x] : \check{\tau}$ is provable* (6.4).

Proof: The proof is by induction on the structure of $\check{a}$ (*i.e.* a, c, b and d). Using lemma 24, we can assume that a derivation of (6.3) does not end with rule Sub.

In each case, we consider a derivation of (6.3). By using a renaming substitution on (6.2) if necessary (proposition 21), we can assume that free variables of τ' that are not in $A^\star$ do not appear free in this derivation (6.5). We write A_x for $A + x : \mathtt{Gen}(\tau', A^\star)$.

We only show the more complicated cases. Other cases are either similar or simple.

Cas a is $\mathtt{let}\ x' = a_1\ \mathtt{in}\ a_2$: A derivation of (6.3) ends as:

$$\frac{A_x \vdash a_1 : \tau_1 \qquad A_x + x_1 : \mathtt{Gen}(\tau_1, A_x) \vdash a_2 : \tau \ \ (6.7)}{A_x \vdash \mathtt{let}\ x_1 = a_1\ \mathtt{in}\ a_2 : \tau}\ (6.6)\ (\textsc{Let})$$

By induction hypothesis applied to (6.6), we get $A \vdash a_1[a'/x] : \tau_1$ (6.8).

If $x_1 = x$, (6.7) becomes $A + x : \mathtt{Gen}(\tau_1, A_x) \vdash a_2 : \tau$. By strengthening of environment (proposition 23), we have $A + x : \mathtt{Gen}(\tau_1, A) \vdash a_2 : \tau$ since A is a subsequence of A_x. We conclude by rule Let.

Otherwise, let A_1 be $A + x_1 : \mathtt{Gen}(\tau_1, A)$. Re-ordering hypotheses in (6.7), we have $A + x_1 : \mathtt{Gen}(\tau_1, A_x) + x : \mathtt{Gen}(\tau', A) \vdash a_2 : \tau$. By strengthening of environment, we can replace A_x by A. Since free type variables of A_1 are the same as free type variables of A, we can replace A by A_1 in $\mathtt{Gen}(\tau', A)$. Thus, we have $A_1 + x : \mathtt{Gen}(\tau', A_1) \vdash a_2 : \tau$. On the other hand, since x_1 is not bound in a', and $A_1^\star$ extends $A^\star$, we deduce $A_1^\star \vdash a' : \tau'$ from (6.2) by extension of environment (proposition 22). Thus, we can apply the induction hypothesis with A_1 for A. We get $A_1 \vdash a_2[a'/x] : \tau$. Combining with (6.8) in a Let rule, we finally have $A \vdash (\mathtt{let}\ x_1 = a_1\ \mathtt{in}\ a_2)[a'/x] : \tau$.

Cas a is $\mathtt{fun}\ (x_1)\ a_2$: A derivation of (6.3) ends as:

$$\frac{A_x + x_1 : \tau_1 \vdash a_2 : \tau_2}{A_x \vdash \mathtt{fun}\ (x_1)\ a_2 : \tau_1 \to \tau_2}\ (\textsc{Fun})$$

Let A_1 be $A + x_1 : \tau_1$. Re-ordering type environment of the premise, we have $A + x_1 : \tau_1 + x : \mathtt{Gen}(\tau', A) \vdash a_2 : \tau_2$. By (6.5), the generalization $\mathtt{Gen}(\tau', A)$ is equal to $\mathtt{Gen}(\tau', A + x_1 : \tau_1)$, that is, $\mathtt{Gen}(\tau', A_1)$. So, we have $A_1 + x : \mathtt{Gen}(\tau', A_1) \vdash a_2 : \tau_2$. Since x_1 is not bound in a' and $A_1^\star$ extends $A^\star$, we deduce $A_1^\star \vdash a' : \tau'$ from (6.2). Thus, we can apply the induction hypothesis with A_1 for A. We get $A_1 \vdash a_2[a'/x] : \tau_2$. We conclude with rule Fun

Cas a is $\langle b \rangle$: A derivation of (6.3) ends as:

$$\frac{A_x^\star + \mathtt{self} : \tau_y \vdash b : \varphi \qquad \tau_y = \langle \mathtt{method}\ (\varphi) \rangle}{A_x \vdash \langle b \rangle : \tau_y}\ (\textsc{Object})$$

Let A_y be $A^\star + \mathtt{self} : \tau_y$. Re-ordering type environment of the premise, we have $A^\star + \mathtt{self} : \tau_y + x : \mathtt{Gen}(\tau', A) \vdash b : \varphi$. We can replace $\mathtt{Gen}(\tau', A)$ by $\mathtt{Gen}(\tau', A^\star)$ by strengthening of environment. By (6.5), the generalization $\mathtt{Gen}(\tau', A^\star)$ is equal to $\mathtt{Gen}(\tau', A^\star + \mathtt{self} : \tau_y)$, that is, $\mathtt{Gen}(\tau', A_y)$. Thus, we have $A_y + x : \mathtt{Gen}(\tau', A_y) \vdash b : \varphi$. Since $A_y^\star$ is just $A^\star$, we have $A_y^\star \vdash a' : \tau'$ (6.3). Thus, we can apply the induction hypothesis with A_y for A. We get $A_y \vdash b[a'/x] : \varphi$. We conclude with rule Object.

■

Lemma 28 (Term replacement (instance variables and `self`)) *Let A be an environment and $\check{a}$ be either an expression a or a class expression c. Let w be an object body and φ be an object body type. We defines U as the restriction of $\mathrm{dom}(w)$ to fields. We write τ_y for $\langle \texttt{method}\ (\varphi)\rangle$. We assume that $A^\star$ is A, bound variables of $\check{a}$ are not be free in $\langle w\rangle$ and $w(u)$, and the following three judgments hold:*

$$A + \texttt{self} : \tau_y \vdash w : \varphi, \qquad (A \vdash w(u) : \tau_u)^{u\in U}, \qquad A + \texttt{self} : \tau_y + (\varphi \setminus \texttt{method}) \vdash \check{a} : \check{\tau}\ (6.9).$$

Then, $A \vdash \check{a}[\langle w\rangle/\texttt{self}][w(u)/u]^{u\in U}[\langle w \mathbin{@} (\texttt{field}\ u = {a_u}^{u\in V})\rangle/\{\langle u = {a_u}^{u\in V}\rangle\}]^{V\subset U} : \check{\tau}$.

Proof: The proof is by induction on the structure of $\check{a}$. For any expression a, we write a^+ for

$$a[\langle w\rangle/\texttt{self}][w(u)/u]^{u\in U}[\langle w \mathbin{@} (\texttt{field}\ u = {a_u}^{u\in V})\rangle/\{\langle u = {a_u}^{u\in V}\rangle\}]^{V\subset U}$$

Class expression c^+ is defined likewise. We write A_y for $A + \texttt{self} : \tau_y + (\varphi \setminus \texttt{method})$. Using lemma 24, we can assume that a derivation of (6.9) does not end with rule Sub.

We only show the more complicated cases. Other cases are easy.

Cas a is `self`: Hypothesis (6.9) is $A + \texttt{self} : \tau_y + (\varphi \setminus \texttt{method}) \vdash \texttt{self} : \tau$. So, τ and τ_y are equal. On the other hand, a^+ is equal to $\langle w\rangle$. We conclude by rule Object:

$$\frac{A + \texttt{self} : \tau \vdash w : \varphi \qquad \tau = \langle \texttt{method}\ (\varphi)\rangle}{A \vdash \langle w\rangle : \tau}\ (\textsc{Object})$$

Cas a is $\{\langle u = {a_u}^{u\in V}\rangle\}$: A derivation of (6.9) ends as:

$$\frac{((6.10)\ \ \texttt{field}\ u : \tau_u \in A_y \qquad (6.11)\ \ A_y \vdash a_u : \tau_u)^{u\in V}}{A_y \vdash \{\langle u : {a_u}^{u\in V}\rangle\} : \tau_y}\ (\textsc{Override})$$

So, from (6.10), $\varphi \oplus \texttt{field}\ u : {\tau_u}^{u\in V} = \varphi$. By induction hypothesis applied to (6.11), we get $A \vdash a_u^+ : \tau_u$ (6.12). Hence $A \vdash (\texttt{field}\ u = a_u^+)^{u\in V} : (\texttt{field}\ u : \tau_u)^{u\in V}$. Then, the append lemma 26 applied to the hypothesis $A + \texttt{self} : \tau_y \vdash w : \varphi$ and the last judgment yields $A + \texttt{self} : \tau_y \vdash w \mathbin{@} (\texttt{field}\ u = a_u^+)^{u\in V} : \varphi$. Hence the following derivation :

$$\frac{A + \texttt{self} : \tau_y \vdash w \mathbin{@} (\texttt{field}\ u = a_u^+)^{u\in V} : \varphi \qquad \tau_y = \langle \texttt{method}\ (\varphi)\rangle}{A \vdash \langle w \mathbin{@} (\texttt{field}\ u = a_u^+)^{u\in V}\rangle : \tau_y}\ (\textsc{Object})$$

■

Lemma 29 (Term replacement (`super`)) *If $A \vdash b_1 : \varphi_1$, $A + \texttt{super}: \varphi \vdash b_2 : \varphi_2$ and bound variables of b_2 are not free in b_1, then $A \vdash b_2' : \varphi_2$ where b_2' is $[a/s\#m]^{\texttt{method}m=a\in b_1}$,* i.e. *$b_2$ where all invocations of methods to super $s\#m$ have been replaced by the body a of the corresponding method m in b_1.*

Proof: The proof is similar to the one of lemma 27. It is in fact simpler, as `super` is not substituted across class and object boundaries, nor across instance variable definitions. ■

Lemma 30 (Redex contraction) *We write $\longrightarrow_\epsilon$ for a one-step reduction in an empty context. If $\check{a}_1 \longrightarrow_\epsilon \check{a}_2$ then $\check{a}_1 \subset \check{a}_2$.*

Proof: The proof is done independently for each redex. All cases are easy now that we have proven the right lemmas.

Let us assume $A \vdash a_1 : \tau$ (6.13) and A equals $A^\star$ (resp. $A \vdash b_1 : \varphi$ (6.14) for any A). We show that $A \vdash a_2 : \tau$ (6.15) (resp. $A \vdash b_2 : \varphi$) by cases on the redex a_1 (resp. b_1). Each case is shown independently. Using lemma 24, we can assume that a derivation of (6.13) does not end with rule Sub.

Cas a_1 is $(\texttt{fun}\ (x)\ a)\ v$: A derivation of (6.13) ends either as:

$$\dfrac{\dfrac{\dfrac{A + x : \tau' \vdash a : \tau_0}{A \vdash \texttt{fun}\ (x)\ a : \tau' \to \tau_0}\ (\textsc{Fun}) \qquad \tau' \to \tau_0 \leq \tau_0' \to \tau}{A \vdash \texttt{fun}\ (x)\ a : \tau_0' \to \tau}\ (\textsc{Sub}) \qquad A \vdash v : \tau_0'}{A \vdash (\texttt{fun}\ (x)\ a)\ v : \tau}\ (\textsc{App})$$

or as:

$$\dfrac{\dfrac{(6.16)\ \ A + x : \tau' \vdash a : \tau}{A \vdash \texttt{fun}\ (x)\ a : \tau' \to \tau}\ (\textsc{Fun}) \qquad (6.17)\ \ A \vdash v : \tau'}{A \vdash (\texttt{fun}\ (x)\ a)\ v : \tau}\ (\textsc{App})$$

The end of the first derivation can be rewritten as:

$$\dfrac{\dfrac{\dfrac{A + x : \tau' \vdash a : \tau_0 \qquad \tau_0 \leq \tau}{(6.16)\ \ A + x : \tau' \vdash a : \tau}\ (\textsc{Sub})}{A \vdash \texttt{fun}\ (x)\ a : \tau' \to \tau}\ (\textsc{Fun}) \qquad \dfrac{A \vdash v : \tau_0' \qquad \tau_0' \leq \tau'}{(6.17)\ \ A \vdash v : \tau'}\ (\textsc{Sub})}{A \vdash (\texttt{fun}\ (x)\ a)\ v : \tau}\ (\textsc{App})$$

In both cases, the term replacement lemma 27 applied to (6.17) and (6.16) shows the conclusion.

Cas c_1 is $(\texttt{fun}\ (x)\ c)\ v$: Similar to previous case.

Cas a_1 is $\texttt{let}\ x = v\ \texttt{in}\ a$: A derivation of (6.13) ends as

$$\dfrac{(6.18)\ \ A \vdash v : \tau' \qquad (6.19)\ \ A + x : \texttt{Gen}(\tau', A) \vdash a : \tau}{A \vdash \texttt{let}\ x = v\ \texttt{in}\ a : \tau}\ (\textsc{Let})$$

The term replacement lemma 27 applied to (6.18) and (6.19) shows the conclusion.

Cas a_1 is $\texttt{class}\ z = v\ \texttt{in}\ a$: Similar to previous case.

Cas a_1 is $\texttt{new}\ (\texttt{struct}\ w\ \texttt{end})$: A derivation of (6.13) ends as

$$\dfrac{\dfrac{A^\star + \texttt{self} : \tau_y \vdash w : \varphi}{A \vdash \texttt{struct}\ w\ \texttt{end} : \texttt{sig}\ (\tau_y)\ \varphi\ \texttt{end}}\ (\textsc{Class-Body}) \qquad \tau_y = \langle \texttt{method}\ (\varphi) \rangle}{(6.20)\ \ A \vdash \texttt{new}\ (\texttt{struct}\ w\ \texttt{end}) : \tau_y}\ (\textsc{New})$$

Hence,

$$\dfrac{A^\star + \texttt{self} : \tau_y \vdash w : \varphi \qquad \tau_y = \langle \texttt{method}\ (\varphi) \rangle}{A \vdash \langle w \rangle : \tau_y}\ (\textsc{Object})$$

Cas a_1 is $\langle w \rangle \# m$: We must remember that $A^\star$ is A. A derivation of (6.13) ends either as

$$\dfrac{\dfrac{\dfrac{A + \texttt{self} : \tau_y \vdash w : \varphi \qquad \tau_y = \langle \texttt{method}\ (\varphi) \rangle}{(\textsc{Object})} \qquad \tau_y \leq \tau_y'}{(\textsc{Sub})\ \ A \vdash \langle w \rangle : \tau_y} \qquad \tau_y' = \langle m : \tau_k'; \tau' \rangle}{A \vdash \langle w \rangle \# m : \tau_k'}\ (\textsc{Send})$$

or as

$$\dfrac{\dfrac{(6.21)\ \ A + \texttt{self} : \tau_y \vdash w : \varphi \qquad (6.22)\ \ \tau_y = \langle \texttt{method}\ (\varphi) \rangle}{(\textsc{Object})} \qquad (6.23)\ \ \tau_y = \langle m : \tau_k; \tau \rangle}{A \vdash \langle w \rangle \# m : \tau_k}\ (\textsc{Send})$$

The end of the first derivation can be rewritten

$$(\textsc{Object})\ \frac{A + \texttt{self} : \tau_y \vdash w : \varphi \qquad \tau_y = \langle \texttt{method}\ (\varphi) \rangle}{(\textsc{Send})\ \dfrac{A \vdash \langle w \rangle : \tau_y \qquad \tau_y = \langle m : \tau_k ; \tau \rangle}{(\textsc{Sub})\ \dfrac{A \vdash \langle w \rangle \# m : \tau_k \qquad \tau_k \leq \tau'_k}{A \vdash \langle w \rangle \# m : \tau'_k}}}$$

It has been seen at the beginning of the proof that rule Sub at the end of a derivation could be ignored. Thus, only the second case need to be considered.

The result is then proved using the term replacement lemma 28.

We first show that the hypotheses of lemma 28 are satisfied. As the fields of an object are typed in the same environment as the object, for $\texttt{field}\ u : \tau_u \in \varphi$, $A \vdash v_u : \tau_u$ (6.24) where $\texttt{field}\ u = v_u \in w$. From (6.22) and (6.23), $\texttt{method}\ m : \tau_k \in \varphi$. Then, from (6.21), an easy induction on w using rules Then, Field, and Method yields:

$$A + \texttt{self} : \tau_y + \varphi_1 \vdash w(m) : \tau_k \text{ for some } \varphi_1 \subset (\varphi \setminus \texttt{method})$$

As A contains no `field` bindings, the environment can be extended to include $\varphi \setminus \texttt{method}$:

$$(6.25)\ \ A + \texttt{self} : \tau_y + (\varphi \setminus \texttt{method}) \vdash w(m) : \tau_k$$

Finally, the term replacement lemma 28 applied to (6.21), (6.24), (6.25) yields

$$A \vdash w(m)[\langle w \rangle / \texttt{self}][w(u)/u]^{u \in U}[\langle w \mathbin{@} (\texttt{field}\ u = {a_u}^{u \in V}) \rangle / \{\langle u = {a_u}^{u \in V} \rangle\}]^{V \subset U} : \tau_k$$

Cas b_1 is `inherit` (`struct` w `end`) `as` s ; b: A derivation of (6.14) ends as

$$\frac{\begin{array}{c}\vdots \\ A \vdash \texttt{inherit}\ (\texttt{struct}\ w\ \texttt{end})\ \texttt{as}\ s : \varphi\end{array} \qquad (6.26)\ \ A + (\varphi \setminus \texttt{method}) \vdash b : \varphi_2}{A \vdash \texttt{inherit}\ (\texttt{struct}\ w\ \texttt{end})\ \texttt{as}\ s\,;b : \varphi_1 \oplus \varphi_2}\ (\textsc{Then})$$

where $\varphi = \varphi_1 + (\texttt{super}\ s : \varphi_1)$, continued by

$$\frac{(6.27)\ \ A \vdash \texttt{self} : \tau_y \qquad \dfrac{(6.28)\ \ A^\star + \texttt{self} : \tau_y \vdash w : \varphi_1}{A \vdash \texttt{struct}\ w\ \texttt{end} : \texttt{sig}\ (\tau_y)\ \varphi_1\ \texttt{end}}\ (\textsc{Class-Body})}{A \vdash \texttt{inherit}\ (\texttt{struct}\ w\ \texttt{end})\ \texttt{as}\ s : \varphi_1 + (\texttt{super}\ s : \varphi_1)}\ (\textsc{Inherit})$$

According to (6.27), $\texttt{self} : \tau_y \in A$. Judgment (6.28) can thus be rewritten $A \vdash w : \varphi_1$ (6.29).

Applying the term replacement lemma 29 on $A + (\varphi_1 \setminus \texttt{method}) \vdash w : \varphi_1$ (the environment has been extended) and (6.26) yields $A + (\varphi_1 \setminus \texttt{method}) \vdash b[a/s\#m]^{\texttt{method}\, m = a \in w} : \varphi_2$. Then, the append lemma applied on (6.29) and this last judgment gives the result:

$$A \vdash w \mathbin{@} b[a/s\#m]^{\texttt{method}\, m = a \in w} : \varphi_1 \oplus \varphi_2$$

Cas b_1 is `field` $u = v$; b: A derivation of (6.14) ends as

$$\frac{\dfrac{A^\star \vdash v : \tau}{A \vdash \texttt{field}\ u = v : (\texttt{field}\ u : \tau)}\ (\textsc{Field}) \qquad (6.30)\ \ A + (\texttt{field}\ u : \tau) \vdash w : \varphi}{A \vdash \texttt{field}\ u = v\,;w : \varphi \oplus (\texttt{field}\ u : \tau)}\ (\textsc{Then})$$

From (6.30), since $u \in dom(w)$ and fields appear before methods in w, an easy induction shows that $A \vdash w : \varphi$. Indeed, fields are typed in environment $A^\star$, and methods are typed in an environment in which $(\texttt{field}\ u : \tau)$ has been added anyway after the typing of the field u appearing in w.

Cas b_1 is $\mathtt{method}\, m = a\,; b$: A derivation of (6.14) ends as

$$\dfrac{\dfrac{A \vdash \mathtt{self} : \langle m : \tau; \tau' \rangle \qquad A \vdash a : \tau}{A \vdash \mathtt{method}\, m = a : (\mathtt{method}\, m : \tau)}\ (\textsc{Method}) \qquad (6.31)\ \ A \vdash w : \varphi}{A \vdash \mathtt{method}\, m = a\,; w : (\mathtt{method}\, m : \tau) \oplus \varphi}\ (\textsc{Then})$$

Since $m \in dom(w)$, $m \in dom(\varphi)$, then φ and $(\mathtt{method}\ m : \tau) \oplus \varphi$ are equal. Therefore, judgment (6.31) can be rewritten $A \vdash w : (\mathtt{method}\, m : \tau) \oplus \varphi$.

Cas a_1 is $(v : \tau <: \tau')$: A derivation of (6.13) ends as

$$\dfrac{A \vdash v : \theta(\tau) \qquad \tau \leq \tau'}{A \vdash (v : \tau <: \tau') : \theta(\tau')}\ (\textsc{Coerce})$$

Hence,

$$\dfrac{A \vdash v : \theta(\tau) \qquad \theta(\tau) \leq \theta(\tau')}{A \vdash v : \theta(\tau')}\ (\textsc{Sub})$$

■

The normal-form theorem is proved by structural induction on values, using the following lemma.

Lemma 31 *Let v be a value. We assume $\emptyset \vdash v : \tau$ (6.32).*

- *If τ is a functional type, then v is a function.*
- *If τ is an object type, then v is an object.*

Let v_c be a value. We assume $\emptyset \vdash v_c : \gamma$.

- *If γ is a functional type, then v is a function.*
- *Otherwise, v is an object.*

Proof: We prove that if v is a function, then τ is a functional type and that if v is an object, then τ is an object type. Then, since a value is either a function or an object and functional types and object types are incompatible, this proves the lemma.

We can ignore rule Sub at the end of a derivation, as it does not change the shape of a type.

Cas a is $\mathtt{fun}\,(x)\, a_1$: A derivation of (6.32) ends as

$$\dfrac{A + x : \tau_1 \vdash a_1 : \tau_2}{A \vdash \mathtt{fun}\,(x)\, a_1 : \tau_1 \to \tau_2}\ (\textsc{Fun})$$

So, τ is $\tau_1 \to \tau_2$.

Cas a is $\langle w \rangle$: A derivation of (6.32) ends as

$$\frac{A^\star + \texttt{self} : \tau_y \vdash w : \varphi \qquad \tau_y = \langle \texttt{method}\ (\varphi) \rangle}{A \vdash \langle w \rangle : \tau_y}\ (\textsc{Object})$$

So, τ is $\langle \texttt{method}\ (\varphi) \rangle$.

The proof is similar for class values.

■

Theorem 2 (Normal forms) *Well-typed irreducible normal forms are values* (i.e. if $\emptyset \vdash a : \tau$ and a cannot be reduced, then a is a value.)

Proof: The proof is by structural induction simultaneously on expressions a and class bodies b. Let us assume $\emptyset \vdash a : \tau$ (6.33) (resp. $\emptyset \vdash c : \gamma$ (6.34), $A \vdash b : \varphi$ (6.35) or $A \vdash d : \varphi$, where A contains only `field` and `method` bindings), and that a (resp. c, b or d) cannot be reduced.

Cas a is x: This expression cannot be typed in the empty environment.

Cas a is $a_1\ a_2$: It is not possible. A derivation of (6.33) shows that there exists a type τ_1 such that $\emptyset \vdash a_1 : \tau_1 \to \tau$. The induction hypothesis applied to expression a_1 shows that it is a value. Since it has a functional type, it must be a function $\texttt{fun}\ (x)\ a_0$. But then expression a could be reduced.

Cas a is $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$: It is not possible. The induction hypothesis applied to expression a_1 shows that it is a value. But then expression a could be reduced.

Cas a is $a_1 \# m$ or $\texttt{class}\ z = c\ \texttt{in}\ a_1$: Similar to previous cases.

Cas a is $\texttt{fun}\ (x)\ a_1$: By definition, expression a is a value.

Cas a is $s \# m$: It is not possible : expression $s \# m$ is not typable in the empty environment.

Cas a is `self` or u or $\{\langle u = {a_u}^{u \in V} \rangle\}$: Same as previous case.

Cas a is $(a_1 : \tau <: \tau')$: It is not possible: a can be reduced.

Cas a is $\langle b \rangle$: The induction hypothesis shows that object body b is a value. Then, expression a is also a value.

Cas a is $\texttt{new}\ c$: It is not possible. A derivation of (6.33) shows that $\emptyset \vdash c : \texttt{sig}\ (\tau_y)\ \varphi\ \texttt{end}$. The induction hypothesis applied to c shows that it is a value. According to its type, it is a structure. But then a can be reduced

Cas c is z: This expression is not typable in the empty environment.

Cas c is $c_1\ a$: It is not possible. A derivation of (6.34) shows that there exists a type τ such that $\emptyset \vdash c_1 : \tau \to \gamma$. The induction hypothesis applied to expression c_1 shows that it is a class value. Since it has a functional type, it must be a function $\texttt{fun}\ (x)\ c_0$. But then expression c could be reduced.

Cas c is $\texttt{fun}\ (x)\ c_1$: By definition, expression c is a value.

Cas c is $\texttt{struct}\ b\ \texttt{end}$: The induction hypothesis shows that class body b is a value. Then, expression c is also a value.

Cas b is $d; b_1$: The induction hypothesis shows that object component d and object body b_1 are in normal forms. d is thus a field or method definition, and it is not overridden by b_1 (otherwise, b could be reduced.)

Cas b is $\emptyset$: By definition, object body b is a value.

Cas d is `inherit` c `as` s: It is not possible. A derivation of (6.35) ends as:

$$\frac{A \vdash \texttt{self} : \tau_y \qquad (6.36)\ \ A \vdash c : \texttt{sig}\ (\tau_y)\ \varphi_1\ \texttt{end}}{A \vdash \texttt{inherit}\ c\ \texttt{as}\ s : \varphi_1 + (\texttt{super}\ s : \varphi_1)}\ (\textsc{Inherit})$$

The induction hypothesis applied to c shows that it is a class value. According to its type, it is of the form `struct` w `end`. But then, the inheritance clause could be reduced.

Cas d is `method` $m = a$: By definition, expression d is in normal form.

Cas d is `field` $u = a$: If $A \vdash d : \texttt{field}\ u : \tau$, then $\emptyset \vdash a : \tau$, as A contains only `field` and `method` bindings. By induction hypothesis, expression a is in normal form. Then, so is object component d.

■

Chapter 7

Poly ML: une extension de ML avec du polymorphisme d'ordre supérieur.

Ce chapitre, publié dans [46], est le résultat d'un travail en collaboration avec Jacques Garrigue.

Extension de ML avec du polymorphisme d'ordre supérieur semi-implicite

Nous proposons une extension modeste et conservatrice de ML qui autorise l'utilisation du polymorphisme d'ordre supérieur de façon semi-implicite. L'introduction des types polymorphes reste entièrement explicite, c'est-à-dire que leur introduction et leur valeur exacte doivent simultanément être indiquées. En revanche, leur élimination est semi-explicite : il suffit d'indiquer leur élimination et le type polymorphe est lui-même synthétisé. Cette extension est particulièrement utile dans le langage Objective ML qui utilise le polymorphisme de façon essentielle et souvent à la place du sous-typage.

Extending ML with Semi-Explicit Higher-Order Polymorphism

We propose a modest conservative extension to ML that allows semi-explicit first-class polymorphism while preserving the essential properties of type inference. In our proposal, the introduction of polymorphic types is fully explicit, that is, both introduction points and exact polymorphic types are to be specified. However, the elimination of polymorphic types is semi-implicit: only elimination points are to be specified as polymorphic types themselves are inferred. This extension is particularly useful in Objective ML where polymorphism replaces subtyping. Objective ML that sustains polymorphism and neglects subtyping.

Introduction

The success of the ML language is due to its combination of several attractive features. Undoubtedly, the polymorphism of ML [35] —or *polymorphism* à la *ML*— with the type inference it allows, is a major advantage. The ML type system stays in close correspondence with the rules of logic, following the Curry-Howard isomorphism between types and formulas, which provides a simple intuition, and a strong type discipline. Simultaneously, type inference relieves the user from the burden of writing types: an algorithm automatically checks whether the program is well-typed and, if true, returns a principal type.

Many extensions that are based on this simple system have been proposed: polymorphic records, first-class continuations, first-class abstract datatypes, type-classes, overloading, objects, etc. In all these extensions, type inference remains straightforward first-order unification with toplevel polymorphism. This shows the robustness of ML-style type inference.

There are of course cases where one would like to have first-class polymorphism, as in system F. ML allows for polymorphic definitions, but abstractions can only be monomorphic. Traditionally, ML polymorphism is used for definitions of first-class functions such as folding or iteration over a parameterized datatype. Some higher-order functionals require polymorphic functions as arguments. These situations mostly appear in encodings, and occurrences in real programs can usually be solved by using functors of the module language.

This simple picture, which relies on a clear separation between data and functions operating on data, has recently been invalidated by several extensions. For instance, data and methods are packed together inside objects. This decreases the need for polymorphism, since methods can be specialized to the piece of data they are embedded with. However, data transformers such as folding functions remain parametric in the type of the output. For instance, a function `fold` with the ML type $\forall\beta,\alpha.\ \beta\ \texttt{list} \to (\beta \to \alpha \to \alpha) \to \alpha \to \alpha$ should become a method for container objects, of type $\forall\alpha.\ (\tau \to \alpha \to \alpha) \to \alpha \to \alpha$ where τ is the type of the elements of the container. The extension of ML with first-class abstract types [65, 108] also requires first-class polymorphic functions: for instance, an expression such as $\lambda f.\texttt{open}\ x\ \texttt{as}\ y\ \texttt{in}\ f\ y$ can only be typed if the argument f is polymorphic in its argument, so that the abstract representation of y is not revealed outside the scope of the open construct. First-class polymorphism seems to be also useful in Haskell to enable the composition of monads.

First-class polymorphic values have been proposed in [108, 83] based on ideas developed in [65]. After de-sugaring, all these proposals reduce to the same idea of using explicit, mutually inverse introduction and elimination functions to coerce higher-order types into basic, parameterized type symbols and back. Therefore, they all face the same problem: types must be written explicitly, both at the introduction and elimination of polymorphism.

Recent results on the undecidability of type inference for system F [123, 59, 90] do not leave many hopes for finding a good subset of system F that significantly extends ML, moreover with decidable type inference and principal types. Previous attempts to accomplish this task were unsuccessful.

This is not the path we choose here. We do not infer higher-order types and thus avoid higher-order unification, undecidable in general. Furthermore, we maintain the simplicity of the ML type system, following the premise that an extension of ML should not modify the ML polymorphism in its essence, even if it is an extension that actually increases the level of polymorphism.

The original insight of our work is that, although ML polymorphism allows type inference, actual ML programs do already contain a lot of type information. All constants, all constructors,

and all previously defined functions already have known types. This information is only waiting to be used appropriately.

In comparison to previous works, we remove the requirement for type annotations at the elimination of polymorphism by using type inference to propagate explicit type information between different points of the program. In our proposal, tagging values of polymorphic types with type symbols becomes superfluous. A type annotation at the introduction of a polymorphic value is sufficient and can be propagated to the elimination site (following the data-flow view of programs). This makes the handling of such values considerably easier, and reasonably practical for use in a programming language.

In a first section, we present our solution informally, and explain how it simplifies the use of higher-order types in ML. Then, we develop this approach formally, proving all fundamental properties. In a third section, encodings are provided, both for previous formulations of first-class polymorphism, and for system F itself. Section 7.4 shows how our system can be used to provide polymorphic methods for Objective ML, in an almost transparent way. In section 7.5 we discuss how the value-only restriction to polymorphism can be applied here. Lastly, we compare with related works, and conclude. Proofs of main theorems are given in appendix.

7.1 Informal approach

In this section we present our solution informally. We first introduce a naive straightforward proposal. We show that this solution needs to be restricted to avoid higher-order unification. Last, we describe a simple solution that allows for complete type inference.

7.1.1 A naive solution

Naively, ML types can be easily extended with polymorphic types. A typical program that cannot be typed in ML and could be typed in system F is $\lambda f.\, ff$. This expression is not very interesting for itself. However, a few variations are sufficient to illustrate most aspects of type inference in the presence of higher-order types. Useful examples can be found in section 7.4 in addition to those suggested in the introduction.

Although $\lambda f.\, ff$ is not typable in ML, the expression $\texttt{let}\ f = \lambda x.\, x\ \texttt{in}\ f\ f$ is. One can see let-definitions as a special syntax, combined with a special typing rule, for the application $(\lambda f.\, ff)\ (\lambda x.\, x)$. Let us exercise by replacing the LET polymorphic binding by first-class polymorphism. The identity $\lambda x.\, x$ of type $\alpha \to \alpha$ has also type scheme $\forall\alpha.\alpha \to \alpha$. We shall write $[\lambda x.\, x : \forall\alpha.\alpha \to \alpha]$ for the *creation* (or *introduction*) of the polymorphic value $\lambda x.\, x$ with type scheme $\forall\alpha.\alpha \to \alpha$. In order to avoid confusion with ML types, we explicitly coerce $\forall\alpha.\alpha \to \alpha$ to a regular ML type $[\forall\alpha.\alpha \to \alpha]$, adding the type constructor $[\_]$. We call $\forall\alpha.\alpha \to \alpha$ a *polymorphic type* or a *type scheme* and $[\forall\alpha.\alpha \to \alpha]$ a *polytype*.

Let f be the expression $[\lambda x.\, x : \forall\alpha.\alpha \to \alpha]$, which has type $[\forall\alpha.\alpha \to \alpha]$. As any first-class value, f can be passed to other functions, be stored in data-structures, etc. For instance $(f, 1)$ is a pair of type $([\forall\alpha.\alpha \to \alpha] \times \texttt{int})$. A polymorphic function (*i.e.* a polymorphic value that is a function) cannot be applied directly, since it is typed with a polytype, which is incompatible with an arrow type. We must previously *open* (or *eliminate*) the polytype. We introduce a new construct $\langle\_\rangle$ for that purpose. Hence, $\langle f \rangle$ is a function of type an instance of the polymorphic type $\forall\alpha.\alpha \to \alpha$, *i.e.* $\tau \to \tau$ for some type τ. Its principal type is $\alpha \to \alpha$.

The raw expression $\lambda f.\, ff$ is not well typed. It should be passed a polymorphic value as argument, for instance, of type $[\forall\alpha.\alpha \to \alpha]$. Here, we shall introduce polymorphism by a type

constraint on the argument: $\lambda f\colon[\forall\alpha.\alpha\to\alpha].\,\langle f\rangle\ f$. The first occurrence of f in the body is opened to eliminate polymorphism before it is applied. The following definition of g is well-typed

$$g \stackrel{def}{=\!=} \lambda f\colon[\forall\alpha.\alpha\to\alpha].\,\langle f\rangle\ f : [\forall\alpha.\alpha\to\alpha]\to[\forall\alpha.\alpha\to\alpha]$$

So are the two following variants:

$$h \stackrel{def}{=\!=} \lambda f\colon[\forall\alpha.\alpha\to\alpha].\,\langle f\rangle\ \langle f\rangle : [\forall\alpha.\alpha\to\alpha]\to\alpha'\to\alpha'$$
$$k \stackrel{def}{=\!=} \lambda f\colon[\forall\alpha.\alpha\to\alpha].\,[\langle f\rangle\ \langle f\rangle : \forall\alpha.\alpha\to\alpha] : [\forall\alpha.\alpha\to\alpha]\to[\forall\alpha.\alpha\to\alpha]$$

In h, the occurrence of f in the argument position is also opened, so the result type is no longer a polytype. In k, polymorphism is lost as in h, then it is recovered explicitly. Finally, we can apply g to f:

$$(\lambda f\colon[\forall\alpha.\alpha\to\alpha].\,\langle f\rangle\ f)\,[\lambda x.\,x : \forall\alpha.\alpha\to\alpha] : [\forall\alpha.\alpha\to\alpha]$$

More interestingly, the following expression is also well-typed

$$(\lambda u.\,u\,[\lambda x.\,x : \forall\alpha.\alpha\to\alpha])\ (\lambda f\colon[\forall\alpha.\alpha\to\alpha].\,\langle f\rangle\ f) : [\forall\alpha.\alpha\to\alpha]$$

There is no term typable in ML that has the same erasure (untyped λ-term) as this one. Note that no type annotation is needed on u since although u has a polytype as result, it is never opened.

7.1.2 An obvious problem

The examples above mixed type-inference and type-checking (using type-annotations). The obvious problem of type inference in the presence of higher-order types remains to be solved: what happens when expressions of unknown type are opened. Should the program $\lambda f.\,\langle f\rangle\ f$ or simpler $\lambda x.\,\langle x\rangle$ be typed?

The answer is clearly negative, since this would amount to inferring higher-order types, which we choose to avoid here. We should keep all user-provided polymorphism, but never guess polymorphism.

The attempt to forbid lambda abstraction of unspecified type to be a polytype does not work. It would violate the assumption that polytypes are regular ML types. Thus, if $\lambda x.\,x$ has type $\alpha\to\alpha$, it should also have type $[\sigma]\to[\sigma]$ for any polymorphic type σ. Actually, it is important that $\lambda x.\,x$ possesses all these types. For instance, both $(\lambda x.\,x)\ f$ and $\lambda x.\,f\ x$ should be typable and have the same type as f.

When typing $\lambda f.\,\langle f\rangle\ f$, variable f is first given an unknown type τ. Guessing $[\forall\alpha.\alpha\to\alpha]$ for τ would be correct, but not principal, since $[\forall\alpha.\alpha\to\alpha\to\alpha]$ would also be a possible type for τ. More subtle, the expression $\lambda f.\,\langle f\rangle\ (g\ f)$ may only be typed with $[\forall\alpha.\alpha\to\alpha]\to[\forall\alpha.\alpha\to\alpha]$ and has a principal derivation. However, we should also reject this program. Informally, type inference would imply backtracking: f is first assumed of unknown type τ; we cannot type $\langle f\rangle$ so we backtrack; typing the application $g\ f$ forces f to be of type $[\forall\alpha.\alpha\to\alpha]$, then $\langle f\rangle$ can be typed, and so on. This causes two problems. Firstly, backtracking may lead to a combinatorial explosion of the search space, and we would rather fail in every case where some inference order would fail. Worse, typing constraints may disappear during reduction. Traditionally, this is not a problem since it only allows to infer better types. However, in our case, the removal of polytype constraints will leave some polytypes unspecified and lead to failure. Consequently, we would loose the subject reduction property. For instance, $\lambda f.\,\langle f\rangle\ (g\ f)$ reduces to $\lambda f.\,\langle f\rangle\ f$ but the latter is not typable.

7.1.3 A simple solution

The essence of our proposal is a simple mechanism based on unification that distinguishes polytypes that have been user-provided from those that have just been guessed. Each occurrence of a polytype $[\sigma]$ is labeled with a label ϵ. That is, we write $[\sigma]^\epsilon$ rather than $[\sigma]$. Actually, we keep $[\sigma]$ as an abbreviation for $[\sigma]^\epsilon$ where ϵ is an anonymous label, *i.e.* one that does not appear anywhere else. Intuitively, labels indicate sharing of polytype nodes.

The elimination of polymorphism $\langle a \rangle$ is possible whenever a can be typed with $[\sigma]^\epsilon$ where ϵ does not appear anywhere else. Informally, we could just say when a has polytype $[\sigma]$ (since ϵ is anonymous). The intuition is that an anonymous label ϵ ensures that the corresponding polytype does not appear anywhere else and *a fortiori* does not appear as an hypothesis (*i.e.* in a negative occurrence, such as the context or the left hand-side of an arrow); thus, it must have been user-provided.

For instance, in the expression $\lambda f.\, \langle f \rangle\ f$, the λ-bound variable f can be given the polytype $[\forall\alpha.\alpha \to \alpha]^\epsilon$, with a monomorphic label ϵ; since all instances of f share the same label ϵ, the label cannot be anonymous as required when typing $\langle f \rangle$. Indeed, the type of the variable f in $\langle f \rangle$ is a polytype only under the assumption that the binding occurrence of f is typed with exactly the same polytype.

On the contrary, when a polytype is explicitly given, it can be propagated top-down. We use polymorphism to generate new anonymous labels from older ones. We allow quantification on anonymous labels, and later instantiation of quantified labels to new anonymous labels.

When typing the expression $\lambda f\!:\![\forall\alpha.\alpha \to \alpha].\, \langle f \rangle\ f$, the type assumption $f : \forall\epsilon.[\forall\alpha.\alpha \to \alpha]^\epsilon$ is added to the context in which $\langle f \rangle\ f$ is typed. Thus, variable f has type $[\forall\alpha.\alpha \to \alpha]^{\epsilon_1}$ with a different, anonymous, label ϵ_1, and therefore $\langle f \rangle$ is well-typed. For technical reasons we chose not to allow type annotation of abstracted variables in our system, but instead $\lambda x\!:\!\tau.\, a$ can be seen as $\lambda x.\, \texttt{let}\ x = (x : \tau)\ \texttt{in}\ a$. Type annotation $(\_ : \tau)$ renames all ϵ's free in τ into fresh ones.

7.2 Formal approach

We formalize our approach as a small extension to core ML.

7.2.1 The core language

Types We assume given two collections of type variables $\alpha \in \mathcal{V}$, and labels $\epsilon \in \mathcal{E}$. The syntax of types is:

$\tau ::= \alpha \mid \tau \to \tau \mid [\sigma]^\epsilon$	Monotypes
$\sigma ::= \tau \mid \forall\alpha.\sigma$	Type schemes
$\varsigma ::= \sigma \mid \forall\epsilon.\varsigma$	Generic schemes
$\xi ::= \alpha \mid \epsilon$	Variables

The construct $[\sigma]^\epsilon$ is used to coerce a type scheme σ to a monotype. We call $[\sigma]^\epsilon$ a weak polytype. The label ϵ is used to keep track of sharing between weak polytypes, or allow them to be usable polytypes, when it is quantified as $\forall\epsilon.[\sigma]^\epsilon$. We do not quantify labels in σ, since this would not add any power to the system (it would be redundant with explicit type annotations).

Free type variables and free labels of a generic scheme, type scheme, or monotype ς are written $FV(\varsigma)$ and $FL(\varsigma)$, and are defined as usual. In a type scheme $\forall\xi.\varsigma$, $\forall$ acts as a quantifier, and the variable or label ξ is bound (*i.e.* not free) in $\forall\xi.\varsigma$. We consider type schemes equal by renaming

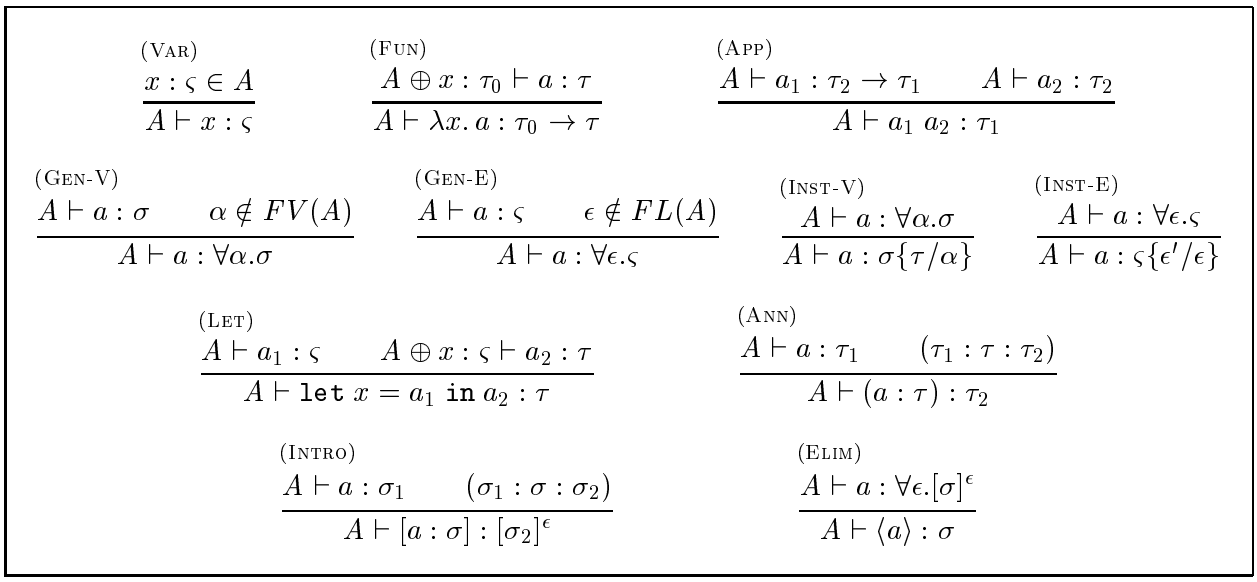


Figure 7.1: Typing rules

and reordering of bound variables and labels, and removal of useless quantifiers (*i.e.* $\forall\xi.\tau \equiv \tau$ whenever variable ξ is not free in τ). As usual, substitutions leave bound variables and bound labels unchanged. For example $(\alpha \to [\forall\beta.\beta \to \alpha]^\epsilon)\{\tau/\alpha\}$ is $\tau \to [\forall\beta.\beta \to \tau]^\epsilon$ provided β is not free in τ. An instance of a type scheme $\forall\bar\epsilon, \bar\alpha.\tau_0$ is $\tau\{\bar\epsilon', \bar\tau/\bar\epsilon, \bar\alpha\}$.

Expressions

$$\begin{aligned} a ::= {} & x \mid \lambda x.\, a \mid a\ a \mid \mathtt{let}\ x = a\ \mathtt{in}\ a \\ & \mid [a : \sigma] \mid \langle a \rangle \mid (a : \tau) \end{aligned}$$

The first line corresponds exactly to core ML. We then introduce three new constructs: introduction and elimination of first-class polymorphism and type annotation.

Typing rules are given in figure 7.1. All typing rules but the last three ones are standard. Rules Ann and Intro use an auxiliary relation $(\_ : \_ : \_)$. Given a type scheme σ, we write $(\sigma_1 : \sigma : \sigma_2)$ if there exists a substitution θ from type variables to types and two substitutions ρ_1 and ρ_2 from labels to labels, such that $\sigma_1 = \theta(\rho_1(\sigma))$ and $\sigma_2 = \theta(\rho_2(\sigma))$. The intuition is that if θ is the identity, then σ_1 and σ_2 are both equal to σ except maybe in their labels. Indeed, $(\rho_1(\sigma) : \sigma : \rho_2(\sigma))$ for any label renamings ρ_1 and ρ_2. If σ does not contain any label, then $(\sigma_1 : \sigma : \sigma_2)$ is equivalent to σ_1 and σ_2 being the same generic instance of σ. An important property of the relation $(\_ : \sigma : \_)$ is its stability by substitution. That is, if $(\sigma_1 : \sigma : \sigma_2)$, then $(\theta(\sigma_1) : \sigma : \theta(\sigma_2))$ for any substitution θ. Note that σ is user-given and is not affected by the substitution.

This relation is used to type explicit annotations. For typechecking purposes, the construct $(\_ : \tau)$ could have been replaced by a countable collection of primitives $\lambda x.\,(x : \tau)$ indexed by τ and given with principal type scheme $\forall\bar\epsilon_1, \bar\epsilon_2, FV(\tau).\ \tau\{\bar\epsilon_1/\bar\epsilon\} \to \tau\{\bar\epsilon_2/\bar\epsilon\}$ where $\bar\epsilon_1$ and $\bar\epsilon_2$ are different renamings of the labels $\bar\epsilon$. That is, to type an expression $(a : \tau)$, let τ_1 and τ_2 be two copies of τ where their labels have been renamed, and θ be a substitution such that a has type $\theta(\tau_1)$; then $(a : \tau)$ has type $\theta(\tau_2)$. We kept annotation as a primitive construct because of its dynamics semantics.

Rule INTRO uses the same relation, except that types schemes replace types. To type $[a : \sigma]$, let σ_1 and σ_2 be two copies of σ where labels have been renamed; find a substitution θ such that a has type $\theta(\sigma_1)$ (*i.e.* $\theta(\sigma_1)$ is a generic instance of the principal type scheme of a); then $[a : \sigma]$ has type $[\theta(\sigma_2)]^\epsilon$ for any label ϵ.

Last, rule ELIM says that polymorphism can be used only if the label of the polytype does not occur anywhere else.

As an example, we have the following derivation, where σ abbreviates $\forall\alpha.\alpha \to \alpha$ and A is $f : [\sigma]^{\epsilon_1}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash f : [\sigma]^{\epsilon_1}} \text{(VAR)} \qquad ([\sigma]^{\epsilon_1} : [\sigma]^{\epsilon} : [\sigma]^{\epsilon_2})}{A \vdash (f : [\sigma]^{\epsilon}) : [\sigma]^{\epsilon_2}} \text{(ANN)}}{A \vdash (f : [\sigma]^{\epsilon}) : \forall\epsilon_2.[\sigma]^{\epsilon_2}} \text{(GEN-E)}}{A \vdash \langle f : [\sigma]^{\epsilon}\rangle : \forall\alpha.\alpha \to \alpha} \text{(ELIM)}}{A \vdash \langle f : [\sigma]^{\epsilon}\rangle : [\sigma]^{\epsilon_1} \to [\sigma]^{\epsilon_1}} \text{(INST-V)}}{\vdots \qquad \dfrac{}{A \vdash f : [\sigma]^{\epsilon_1}} \text{(VAR)}}{A \vdash \langle f : [\sigma]^{\epsilon}\rangle\ f : [\sigma]^{\epsilon_1}} \text{(APP)}}{\vdash \lambda f.\, \langle f : [\sigma]^{\epsilon}\rangle\ f : [\sigma]^{\epsilon_1} \to [\sigma]^{\epsilon_1}} \text{(FUN)}$$

$(\sigma \equiv \forall\alpha.\alpha \to \alpha)$

$(\sigma \leftarrow [\sigma]^{\epsilon_1})$

7.2.2 Dynamic semantics

We give a call-by-value semantics for the core language. Values and evaluation contexts are:

$$\begin{aligned} v &::= w \mid [v : \sigma] \\ w &::= \lambda x.\, a \mid (w : \tau_1 \to \tau_2) \\ E &::= \{\} \mid E\ a \mid v\ E \mid \texttt{let}\ x = E\ \texttt{in}\ a \mid [E : \sigma] \mid (E : \tau) \mid \langle E\rangle \end{aligned}$$

One step is either a reduction of the form:

$$\begin{aligned} (\lambda x.\, a)\ v &\xrightarrow{\mathsf{Fun}} a\{v/x\} \\ \texttt{let}\ x = v\ \texttt{in}\ a &\xrightarrow{\mathsf{Let}} a\{v/x\} \\ \langle [v : \forall\bar{\alpha}.\tau]\rangle &\xrightarrow{\mathsf{Elim}} (v : \tau) \\ (v_1 : \tau_2 \to \tau_1)\ v_2 &\xrightarrow{\mathsf{Tfun}} (v_1\ (v_2 : \tau_2) : \tau_1) \\ ([v : \forall\bar{\alpha}.\tau] : [\sigma]^{\epsilon}) &\xrightarrow{\mathsf{Tint}} [(v : \tau) : \sigma] \\ (v : \alpha) &\xrightarrow{\mathsf{Tvar}} v \end{aligned}$$

or an inner reduction obtained by induction:

$$\frac{a_1 \xrightarrow{\mathsf{r}} a_2}{E\{a_1\} \xrightarrow{\mathsf{r}} E\{a_2\}}$$

Note that α, in rule TVAR, is really a variable and not a meta-variable. It is a major difference with ML that type annotations are not just a means to restrict principal types to instances. On the opposite, they allow better typings. Thus, reduction must preserve type annotations as long as they provide useful typing information. Indeed, while terms are only reduced by rules FUN, LET, and ELIM, we need the rules TFUN and TINT to maintain this type information. Rule TVAR erases empty type information. Although types are preserved during reduction, they do not actually

participate in the reduction. In particular, it would be immediate to define an untyped reduction $\xrightarrow{u}$ and a type-erasure er, and to show that if $a_1 \longrightarrow a_2$, then $er(a_1) \xrightarrow{u} er(a_2)$ or $er(a_1)$ and $er(a_2)$ are equal.

7.2.3 Type soundness

We could easily show that evaluation cannot go wrong by means of translation into system F. We prefer to prove it in a more direct way. Subject reduction is an intermediate result of the direct proof that is neither required nor implied by type soundness. However, it is quite important for itself, since it shows that each reduction step preserves typings, and thus that the static semantics is tightly related to the dynamic semantics.

Subject reduction is not obviously preserved by extension to polytypes: the new constructions allow more programs to be typed, but simultaneously, their reduced forms need more programs to be typable. In particular, subject reduction would not hold if we threw away type constraints too early during reduction.

Both subject reduction and type inference are simplified by restricting ourselves to canonical derivations. A similar result existed for the original Damas-Milner presentation of ML, but ML is now often presented in its syntax directed form.

Canonical derivations are those where occurrences of rules GEN and INST are restricted as follows:

- rule GEN only occurs as the last rule of the derivation or right above rule INTRO, ELIM, the left premise of rule LET, or another rule GEN.
- rule INST may only occur right after rule VAR, rule ELIM, or another rule INST.

Lemma 32 (Canonical derivations) *A valid typing judgment $A \vdash a : \tau$ has a canonical derivation.*

Another classical result is the stability of typing judgments by substitution:

Lemma 33 (Stability) *If $A \vdash a : \tau$, then for any substitution θ, $\theta(A) \vdash a : \theta(\tau)$.*

It is important to notice that the substitution is not applied to the expression a, in particular type constraints inside a are left unchanged: their free variables must be understood as if they were closed by existential quantification.

We define a relation $a_1 \subset a_2$ between programs stating that all typings of a_1 are also typings of a_2, *i.e.*

$$a_1 \subset a_2 \stackrel{def}{=\!=} (\forall A, \varsigma,\ A \vdash a_1 : \varsigma \Longrightarrow A \vdash a_2 : \varsigma)$$

Theorem 3 (Subject reduction) *Reduction preserves typings,* i.e. *if $a_1 \longrightarrow a_2$, then $a_1 \subset a_2$.*

Subject reduction is not sufficient to prove type soundness, since the full relation (every program has every type in any context) satisfies subject reduction but does not prevent from type errors. It must be complemented by the following result:

Theorem 4 (Canonical forms) *Irreducible programs that are well-typed in the empty environment are values.*

Type soundness is a straightforward combination of the two previous theorems.

7.2.4 Type inference

First-order unification on simple types must be extended to handle polytypes. During unification, a polytype is treated as a rigid skeleton corresponding to the polymorphic part, on which hang simple types. We present both unification and type inference as solving unification constraints, following [57]. The formalism used is that of conditional rewriting, where we distinguish between assumed conditions, which can always be satisfied, written **let** ... **in** , and conditions that may fail, providing dynamic control during the inference process, written **if** ... **then**.

Unification for simple types First, we remind unification for simple types. In this part only, we exclude polytypes from types τ. A unification problem is a formula U defined by the following grammar.

$$\begin{array}{ll} U ::= \bot \mid \top \mid U \wedge U \mid \exists\, \alpha.\, U \mid e & \text{Unification problems} \\ e ::= \tau \mid \tau \doteq e & \text{Multi-equations} \end{array}$$

The symbols $\top$ and $\bot$ are respectively the trivial and unsatisfiable unification problems. We treat them as a unit and a zero for $\wedge$. That is $U \wedge \top$ and $U \wedge \bot$ are equal to U and $\bot$, respectively. We also identify $\top$ with singleton multi-equations. That is, we can always consider that a unification problem U contains at least one multi-equation $\alpha \doteq e$ for each variable of U. A complex formula is the conjunction of other formulas or the existential quantification of another formula. The symbol $\wedge$ is commutative and associative.

The symbol $\exists$ will be needed later for polytypes. It acts as a binder, *i.e.* free variables of $\exists\, \alpha.\, U$ are free variables of U except α. Bound variables can freely be renamed. We identify $\exists\, \alpha_1.\, \exists\, \alpha_2.\, U$ and $\exists\, \alpha_2.\, \exists\, \alpha_1.\, U$ and simply write $\exists\, \alpha_1, \alpha_2.\, U$. The symbol $\doteq$ is associative and commutative. This makes multi-equations behave as multi-sets of terms.

The substitution of terms is extended to unificands in a straightforward way. For existentials, the application of a substitution θ to a unificand $\exists\, \alpha.\, U$ is the unificand $\exists\, \alpha'.\, \theta(U\{\alpha'/\alpha\})$ where α' is chosen outside of both the domain and the codomain of θ and outside free variables of U.

A substitution θ is a solution of a multi-equation if it sends all terms of the multi-equation to the same codomain. The substitution θ satisfies a conjunction of subproblems if it satisfies all subproblems; θ is a solution of $\exists\, \alpha.\, U$ if it can be extended on α' into a solution of $U\{\alpha'/\alpha\}$ where α' is chosen outside of both the domain and the codomain of θ and outside free variables of U.

Two unification problems are equivalent if they have the same set of solutions. One can check that all previous structural equalities are indeed equivalences. We write $U_1 \equiv U_2$ when the unification problems U_1 and U_2 are equivalent. We also write $U_1 \Rrightarrow U_2$ to mean that the unification problem U_1 can be rewritten into the equivalent unification problem U_2.

Given a unification problem U, we define the containment ordering $\prec_U$ as the transitive closure of the immediate precedence ordering containing all pairs $\alpha \prec \alpha'$ such that there exists a multi-equation $\alpha \doteq \tau \doteq e$ in U where τ is a non-variable term that contains α'. A unification problem is strict if $\prec_U$ is strict. Remark that strictness is syntactic and is not preserved by equivalence. The detection of cycles by a non strict containment ordering is always sound; it is also complete, but only for fully merged and decomposed unification problems.

A problem is in solved form if it is either $\bot$ or $\top$, or if it is strict, merged, decomposed, and of the form $\exists\, \bar{\alpha}.\ \bigwedge_{i \in 1..n} e_i$. In particular, multi-equations e_i contain at most one non-variable term, and if $i \neq j$ then e_i and e_j contain no variable term in common. An explicit principal solution θ can be read straightforwardly from a problem in solved form. We also write $U \Rrightarrow \exists\, \bar{\xi}.\, \theta$ if θ is a principal solution of U and variables $\bar{\xi}$ are not free in U, or by abuse of notation, if U is unsatisfiable and

OCCUR-CHECK
if $\prec_U$ is not strict **then**
$U \Rrightarrow \bot$
MERGE
$\alpha \doteq e \wedge \alpha \doteq e' \Rrightarrow \alpha \doteq e \doteq e'$
ABSORB
$\alpha \doteq \alpha \doteq e \Rrightarrow \alpha \doteq e$
DECOMPOSE
if $size(\tau_1 \to \tau_2) \leq size(\tau_1' \to \tau_2')$ **then**
$\tau_1 \to \tau_2 \doteq \tau_1' \to \tau_2' \doteq e \Rrightarrow \tau_1 \to \tau_2 \doteq e \wedge \tau_1 \doteq \tau_1' \wedge \tau_2 \doteq \tau_2'$

Figure 7.2: First-order unification for simple types

θ is $\bot$. This is consistent with the previous notation since θ could be seen as $\wedge_{\alpha \in dom(\theta)} \alpha \doteq \theta(\alpha)$ whenever its domain and codomain are disjoint.

The unification algorithm is given as a set of rewriting rules that preserve equivalence in figure 7.2. There are implicit context rules that allow to rewrite complex formulas by rewriting any sub-formula. We write $size(\sigma)$ the size of term σ counted as the number of occurrences of symbols $(\_ \to \_)$ or $[\_]$ in σ. These rules are all standard. It is well-known that given an arbitrary unification problem, applying these rules always terminate with a unification problem in solved-formed. The rule OCCUR-CHECK rejects solutions with recursive types. If it were omitted the algorithm would infer recursive types.

Unification for simple-types with polytypes We now allow polytypes $[\sigma]^\epsilon$. In order to allow a natural decomposition of polytypes, we extend typing problems with equations between type schemes.

$$U ::= \ldots \mid \sigma \doteq \sigma$$

These are not multi-equations. In particular, a variable cannot be equated to a polymorphic type scheme, and as a result, equations involving type schemes are never merged.

A substitution θ is a solution of a polytype equation $\forall \bar{\alpha}.\tau \doteq \forall \bar{\alpha}'.\tau'$ (1) if $\theta(\forall \bar{\alpha}.\tau) = \theta(\forall \bar{\alpha}'.\tau')$, where equality is the usual equality for type schemes in ML, *i.e.* it is taken modulo reordering and renaming of universal quantifiers, and removal of useless universal variables. This is equivalent to the existence of two injective substitutions ρ and ρ' of respective domain $\bar{\alpha}$ and $\bar{\alpha}'$ and of codomain $\bar{\alpha}\bar{\alpha}'$, a renaming η from $\bar{\alpha}\bar{\alpha}'$ outside of free variables of θ, τ, τ', and $\bar{\alpha}\bar{\alpha}'$ such that $\theta \circ \eta$ is a solution of $\rho(\tau) = \rho'(\tau')$. We could solve such unification problems by first unifying $\rho(\tau)$ and $\rho'(\tau')$ and then checking the constraints. However, this would force some unnecessary dependence. Intuitively, the renaming η can be dealt with by existential quantification of unificands. In particular, η can be the identity when θ is disjoint from $\bar{\alpha}\bar{\alpha}'$.

Without loss of generality, we can restrict ourselves to the case where $\bar{\alpha} \cap \bar{\alpha}'$, $FV(\tau) \cap \bar{\alpha}'$, and $FV(\tau') \cap \bar{\alpha}$ are all empty sets (1). Let θ' be $(\eta + \eta^{-1}) \circ \theta \circ \eta \circ (\rho + \rho')$. The substitution θ' also decomposes as $(\eta \circ \theta \setminus \bar{\alpha}\bar{\alpha}') + (\rho + \rho')$. Clearly, it satisfies the three following properties:

1. $\theta'(\tau) = \theta'(\tau')$,
2. $\theta' \restriction \bar{\alpha}$ and $\theta' \restriction \bar{\alpha}'$ are injective in $\bar{\alpha}\bar{\alpha}'$, and

DECOMPOSE-POLY
if $size(\sigma) \leq size(\sigma')$ **then**
$[\sigma]^{\epsilon} \doteq [\sigma']^{\epsilon'} \doteq e \Rrightarrow [\sigma]^{\epsilon} \doteq e \wedge \epsilon \doteq \epsilon' \wedge \sigma \doteq \sigma'$

CLASH
$[\sigma]^{\epsilon} \doteq \tau \to \tau' \doteq e' \Rrightarrow \bot$

POLYTYPES
let $\bar{\alpha} \cap \bar{\alpha}' = \emptyset$ **and** $\bar{\alpha} \cap FV(\tau') = \emptyset$ **and** $\bar{\alpha}' \cap FV(\tau) = \emptyset$ **in**
$\forall \bar{\alpha}.\tau \doteq \forall \bar{\alpha}'.\tau' \Rrightarrow \exists\, \bar{\alpha}\bar{\alpha}'.\, \tau \doteq \tau' \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}'$

RENAMING-TRUE
let $\bar{\alpha} = (\alpha_i)^{i \in 1..n+p}$ **and** $\bar{\alpha}' = (\alpha'_i)^{i \in 1..n+q}$ **in**
$\exists\, \bar{\alpha}\bar{\alpha}'.\, (\alpha_i \doteq \alpha'_i)^{i \in 1..n} \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}' \Rrightarrow \top$

RENAMING-FALSE
if $\beta \in \bar{\alpha}$ **and** $\tau \notin \bar{\alpha}' \cup \{\beta\}$ **then** $\beta \doteq \tau \doteq e \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}' \Rrightarrow \bot$
if $\beta \in \bar{\alpha} \cap FV(\tau)$ **and** $\tau \neq \beta$ **then** $\gamma \doteq \tau \doteq e \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}' \Rrightarrow \bot$

Figure 7.3: First-order unification for simple types with polytypes

3. no variable of $\bar{\alpha}\bar{\alpha}'$ appears in $im\,(\theta' \setminus \bar{\alpha}\bar{\alpha}')$.

Conversely, a substitution θ' satisfying these three conditions is a solution of $\forall \bar{\alpha}.\tau \doteq \forall \alpha'.\tau'$.

Indeed, the condition 1 above is a unification problem. We introduce a new kind of unificands $\bar{\alpha} \leftrightarrow \bar{\alpha}'$ whose solutions are substitutions satisfying the conditions 2 and 3. We consider $\bar{\alpha}$ and $\bar{\alpha}'$ as multi-sets (*i.e.* the comma is associative and commutative). In order to avoid special cases, we also require that no variable is listed twice in the sequence $\bar{\alpha}\bar{\alpha}'$ (in particular $\bar{\alpha} \cap \bar{\alpha}'$ is empty). The symbols $\doteq$ (in polytype equations) and $\leftrightarrow$ are commutative. Then θ is a solution of $\forall \bar{\alpha}.\tau \doteq \forall \bar{\alpha}'.\tau'$ under the assumption (1), if and only if it is a solution $\exists\, \bar{\alpha}\bar{\alpha}'.\, (\tau \doteq \tau' \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}')$. Remark that unificands are no longer stable by arbitrary substitutions as long as they contain free variables appearing in renaming unificands (otherwise, renaming unificands could even become ill-formed.) Still, unificands remain stable by renamings. Indeed this is necessary to give meaning to existentially quantified unificands.

Rules for unification with polytypes are those of figure 7.2 plus those of figure 7.3. Rule CLASH handles type incompatibilities. Rule POLYTYPES transforms polytype equations as described above. Rule RENAMING-TRUE allows to remove a satisfiable renaming constraint that became garbage, *i.e.* independent of all other multi-equations. On the opposite, rule RENAMING-FALSE detects unsolvable renaming constraints. In the first case, a solution θ of $\bar{\alpha} \leftrightarrow \bar{\alpha}'$ would identify a variable β of $\bar{\alpha}$ with another variable of $\bar{\alpha}$ (thus θ would not be injective) or with a term outside of $\bar{\alpha} \cup \bar{\alpha}'$. In the second case, the image of a variable γ would contain properly a variable β of $\bar{\alpha}$, making it leak into a wider environment (thus, violating condition 3).

It can be easily checked that if U is merged and decomposed, then for every renaming constraint that remains either rule RENAMING-TRUE or -FALSE applies. Therefore, renaming constraints can always be eliminated.

Theorem 5 *Given a unification problem U, there exists a most general unifier θ which is computed by the set of rules in figures 7.2 and 7.3, or there is no unifier and the rules reduce to $\bot$.*

VAR
let $\forall\bar{\xi}.\tau' = A(x)$ **and** $\bar{\xi} \cap FV(\tau) = \emptyset$ **in**
$A \triangleright x : \tau \Rrightarrow \exists\,\bar{\xi}.\,\tau \doteq \tau'$

FUN
let $\alpha_1, \alpha_2 \notin FV(A) \cup FV(\tau)$ **in**
$A \triangleright \lambda x.\,a : \tau \Rrightarrow \exists\,\alpha_1, \alpha_2.\,(A \oplus x : \alpha_1 \triangleright a : \alpha_2) \wedge \tau \doteq \alpha_1 \to \alpha_2$

APP
let $\alpha \notin FV(A) \cup FV(\tau)$ **in**
$A \triangleright a_1\ a_2 : \tau \Rrightarrow \exists\,\alpha.\,(A \triangleright a_1 : \alpha \to \tau) \wedge (A \triangleright a_2 : \alpha)$

LET
let $\alpha \notin FV(A)$ **in**
if $A \triangleright a_1 : \alpha \Rrightarrow \exists\,\bar{\xi}.\,\theta$ **then**
$A \triangleright \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau \Rrightarrow \exists\,\bar{\xi}, \alpha.\,\theta \wedge A \oplus x : \mathit{Gen}\,(\theta(\alpha), \theta(A)) \triangleright a_2 : \tau$
else $A \triangleright \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau \Rrightarrow \bot$

ANN
let $\bar{\epsilon}_0 = FL(\tau_0)$ **and** $\bar{\epsilon}_1$ and $\bar{\epsilon}_2$ be disjoint copies of $\bar{\epsilon}_0$ outside of A and τ
and $\bar{\alpha}_0 = FV(\tau_0)$ **and** $\bar{\alpha}_1$ be a copy of $\bar{\alpha}_0$ outside of A and τ
and $\tau_1 = \tau_0\{\bar{\alpha}_1/\bar{\alpha}_0\}$ **in**
$A \triangleright (a : \tau_0) : \tau \Rrightarrow \exists\,\bar{\epsilon}_1, \bar{\epsilon}_2, \bar{\alpha}_1.\,A \triangleright a : \tau_1\{\bar{\epsilon}_1/\bar{\epsilon}_0\} \wedge \tau \doteq \tau_1\{\bar{\epsilon}_2/\bar{\epsilon}_0\}$

INTRO
let $\sigma = \forall\bar{\alpha}.\tau_0$ **and** $\bar{\alpha} \cap FV(A) = \emptyset$
and $\bar{\epsilon}_0 = FL(\sigma)$ **and** $\bar{\epsilon}_1$ and $\bar{\epsilon}_2$ be disjoint copies of $\bar{\epsilon}_0$ outside of A and τ
and $\bar{\alpha}_0 = FV(\sigma)$ **and** $\bar{\alpha}_1$ be a copy of $\bar{\alpha}_0$ outside of A, τ and $\bar{\alpha}$
and $\tau_1 = \tau_0\{\bar{\alpha}_1/\bar{\alpha}_0\}$ **in**
if $A \triangleright a : \tau_1\{\bar{\epsilon}_1/\bar{\epsilon}_0\} \Rrightarrow \exists\,\bar{\xi}.\,\theta$ **and** $\bar{\alpha} \cap (\mathit{dom}\,(\theta) \cup FV(\mathit{im}\,(\theta))) = \emptyset$ **then**
$A \triangleright [a : \sigma] : \tau \Rrightarrow \exists\,\bar{\xi}, \bar{\epsilon}_1, \bar{\epsilon}_2, \bar{\alpha}_1, \epsilon.\,\theta \wedge \tau \doteq [\forall\bar{\alpha}.\tau_1\{\bar{\epsilon}_2/\bar{\epsilon}_0\}]^{\epsilon}$
else $A \triangleright [a : \sigma] : \tau \Rrightarrow \bot$

ELIM
let $\alpha \notin FV(A)$ **in**
if $A \triangleright a : \alpha \Rrightarrow \exists\,\bar{\xi}.\,\theta$ **then**
if $\theta(\alpha) = [\forall\bar{\alpha}'.\tau']^{\epsilon}$ **and** $\epsilon \notin FL(\theta(A))$ **then** $A \triangleright \langle a \rangle : \tau \Rrightarrow \exists\,\bar{\xi}, \alpha, \bar{\alpha}'.\,\theta \wedge \tau' \doteq \tau$
else if $\theta(\alpha) = \alpha'$ **and** $\alpha' \notin FV(\theta(A))$ **then** $A \triangleright \langle a \rangle : \tau \Rrightarrow \exists\,\bar{\xi}, \alpha.\,\theta$
else $A \triangleright \langle a \rangle : \tau \Rrightarrow \bot$
else $A \triangleright \langle a \rangle : \tau \Rrightarrow \bot$

Figure 7.4: Type inference algorithm

Type inference For type inference, we extend atomic formulas with typing problems. A typing problem is a triple, written $A \triangleright a : \tau$, of an environment A, a term a, and a type τ. A solution of a typing problem $A \triangleright a : \tau$ is a substitution θ such that $\theta(A) \vdash a : \theta(\tau)$. By lemma 33, the set of solutions of a typing problem is stable under substitutions. Thus, typing problems can be treated as unification problems, following [102]. The rules for solving typing problems are given in figure 7.4. The generalization $\mathit{Gen}\,(\sigma, A)$ is, as usual, $\forall\bar{\xi}.\sigma$ where $\bar{\xi}$ are all free variables and free labels of σ that do not occur in A. To lighten the presentation we leave it implicit that whenever we write $\exists\,\bar{\xi}.\,\theta$, variables $\bar{\xi}$ are asumed to be distinct from all other variables appearing in the rule.

Theorem 6 *Given a typing problem $(A \triangleright a : \tau)$ there exists a principal solution, which is computed by the set of rules described in figures 7.2, 7.3 and 7.4, or there is no solution and the rules reduce to $\bot$.*

7.2.5 Printing labels as sharing constraints

We propose here an alternative interface to the system, potentially enhancing readability of types shown to the user. It is robust, and could also have been used in the presentation of our type system. We preferred the other, more traditional approach for sake of readability.

Labels are used to trace the sharing of polytypes. Types can be restricted so that two polytypes with the same label are necessarily equal. This property is not required in the present type system, but it is stable: if satisfied by all initial type assumptions in A and type annotations in a, then it remains valid in all types appearing in a principal derivation of $A \vdash a : \tau$. The grammar of types can be extended with a sharing construct[1]:

$$\tau ::= \ldots \mid (\tau \texttt{ where } \alpha = \tau)$$

Using sharing, any type can always be written such that every label occurs at most once, and thus can be omitted. In fact, in our presentation, sharing of types is preserved during type inference. Sharing was just ignored when reading principal solutions from unificands in solved form. The `where` construct allows to read and print all sharing present in the solved form. Actually, only the sharing involving polytypes needs to be printed; the other sharing can be ignored.

For instance, the expression $\lambda x.(x : [\sigma])$ has type $[\sigma] \to [\sigma]$, since the two polytypes have different labels, but the expression $\lambda x.\,\texttt{let } y = (x : [\sigma]) \texttt{ in } x$ has type $(\alpha \to \alpha \texttt{ where } \alpha = [\sigma])$.

Both notations (sharing constraints and label variables) actually coincide when all polytypes are anonymous (*i.e.* no label variable occur twice) and polytypes are simply written $[\sigma]$ for instance $\lambda x.(x : \sigma)$ has type $[\sigma] \to [\sigma]$. This is an important case, since the only types the user actually needs to write are of this form. Indeed, types written by the user are only type annotations, which become more general by removing sharing constraints. More precisely, if σ' is a type scheme obtained from σ by a label substitution ρ, then for any expression a, we have $(a : \sigma) \subset (a : \sigma')$ and $[a : \sigma] \subset [a : \sigma']$. This is an easy consequence of σ being more general than σ'.

Thus, the user never needs to write labels or sharing constraints, but he must read them in both inferred types and type-error messages.

7.3 Encodings

In this section, we give encodings in our language for both system F and explicit polymorphism with datatypes. This last encoding is direct, and makes our language an alternative to system F.

Type annotation on arguments

It is convenient to allow $\lambda x{:}\tau.\,a$ in expressions. We see such expressions as syntactic sugar for $\lambda x.\,\texttt{let } x = (x : \tau) \texttt{ in } a$. The derived typing rule is:

$$\frac{A \oplus (x : \forall FL(\tau_2) \setminus FL(\tau_1).\tau_2) \vdash a : \tau' \qquad (\tau_1 : \tau : \tau_2)}{A \vdash \lambda x{:}\tau.\,a : \tau_1 \to \tau'} \text{(Poly-Fun)}$$

[1] Alternatively, one could use the binding τ `as` α as in Objective ML, although the binding scope of `as` is less clear and harder to deal with, formally.

The derived reduction is $(\lambda x{:}\tau.\, a)\ v \xrightarrow{\mathsf{Fun}} a\{(v\ :\ \tau)/x\}$. Note that τ_1 is not just the result of renaming label variables of τ. It may also be an instance of τ. Hence, the set $FL(\tau_2) \setminus FL(\tau_1)$ contains only labels corresponding to copies of those of τ and do not include any label that would have been brought by the instance of a free type variables of τ (since those would also appear in τ_1).

Polymorphic datatypes

Previous works have used data types to provide explicit polymorphism [65, 108, 83]. Omitting other aspects that are irrelevant here, all these works amount to an extension of ML with expressions of the form:

$$\begin{array}{rll} t ::= \alpha \mid t \to t \mid T\ \bar{\alpha} & & \text{Types} \\ M ::= x \mid M\ M \mid \lambda x.\, M \mid T\ M \mid T^{-1}\ M & & \text{Terms} \\ \quad \mid \mathtt{type}\ T\ \bar{\alpha} = \sigma\ \mathtt{in}\ M & & \text{Type declarations} \end{array}$$

where T ranges over datatype symbols. In expressions, T and T^{-1} act as mutually inverse introduction and elimination functions to coerce the higher-order type σ into the simple type $T\ \bar{\alpha}$.

The translation is an inductive definition $\langle\!\langle \_ \rangle\!\rangle_\rho$. The environment ρ is a list of type definitions $\mathtt{type}\ T\ \bar{\alpha} = \sigma_0$ and $\rho(T)$ is the function $\Lambda\bar{\alpha}.\, \sigma_0$, *i.e.* given type arguments $\bar{\tau}$, it returns the type $\sigma_0\{\bar{\tau}/\bar{\alpha}\}$, using the right most definition of T in ρ. The translation of these types into types of our language is straightforward. The translation does not actually use type annotations smartly, and uses a single label ϵ. It could also make all labels of the translation different, *i.e.* anonymous, but this is not needed.

$$\langle\!\langle \alpha \rangle\!\rangle_\rho = \alpha \qquad \langle\!\langle t_1 \to t_2 \rangle\!\rangle_\rho = \langle\!\langle t_1 \rangle\!\rangle_\rho \to \langle\!\langle t_2 \rangle\!\rangle_\rho \qquad \langle\!\langle T\ \bar{t} \rangle\!\rangle_\rho = [\rho(T)\ \overline{\langle\!\langle t \rangle\!\rangle_\rho}]^\epsilon$$

We translate programs as follows.

$$\langle\!\langle x \rangle\!\rangle_\rho = x \qquad \langle\!\langle \lambda x.\, M \rangle\!\rangle_\rho = \lambda x.\, \langle\!\langle M \rangle\!\rangle_\rho \qquad \langle\!\langle M_1\ M_2 \rangle\!\rangle_\rho = \langle\!\langle M_1 \rangle\!\rangle_\rho\ \langle\!\langle M_2 \rangle\!\rangle_\rho$$

$$\langle\!\langle T\ M \rangle\!\rangle_\rho = [\langle\!\langle M \rangle\!\rangle_\rho : \rho(T)\ \bar{\alpha}] \qquad \langle\!\langle T^{-1}\ M \rangle\!\rangle_\rho = \langle \langle\!\langle M \rangle\!\rangle_\rho : [\rho(T)\ \bar{\alpha}]^\epsilon \rangle$$

$$\langle\!\langle \mathtt{type}\ T\ \bar{\alpha} = t\ \mathtt{in}\ a \rangle\!\rangle_\rho = \langle\!\langle M \rangle\!\rangle_{\rho, \mathtt{type}\ T\ \bar{\alpha} = t}$$

Indeed, the pattern $\langle \_ : [\sigma] \rangle$ amounts to the explicit elimination of polymorphism. Since, in the translation, the elimination of polymorphism is always explicit, it can easily be shown that the translation of a well-typed term is always well-typed. (While the program uses only one label, the type derivation need at least two other labels to locally type the elimination patterns $\langle \langle\!\langle M \rangle\!\rangle_\rho : [\rho(T)\ \bar{\alpha}]^\epsilon \rangle$.)

Encoding system F

Laüfer and Odersky have shown an encoding of system F into polymorphic datatypes [83]. This guarantees by composition that system F can be encoded into semi-explicit polymorphism. We give here a direct encoding of system F, which is much simpler than the encoding into polymorphic datatypes.

The types and the terms of system F are

$$\begin{array}{rll} t ::= \alpha \mid t \to t \mid \forall \alpha.t & & \text{Types} \\ M ::= x \mid M\ M \mid \lambda x{:}\, t.\, M \mid \Lambda\alpha.\, M \mid M\ t & & \text{Terms} \end{array}$$

The translation of types of system F into types of our language is again straightforward, and may use a single label ϵ:

$$\langle\!\langle \alpha \rangle\!\rangle = \alpha \qquad \langle\!\langle t_1 \to t_2 \rangle\!\rangle = \langle\!\langle t_1 \rangle\!\rangle \to \langle\!\langle t_2 \rangle\!\rangle \qquad \langle\!\langle \forall\alpha.t \rangle\!\rangle = [\forall\alpha.\langle\!\langle t \rangle\!\rangle]^\epsilon$$

The translation $\langle\!\langle \_ \rangle\!\rangle$ is extended to typing environments in an homomorphic way. The translation of typing derivations of terms of system F into terms of our language is given by the following inference rules:

$$\frac{x : t \in A}{A \vdash x : t \Rightarrow x} \qquad \frac{A \oplus (x : t) \vdash M : t' \Rightarrow a}{A \vdash \lambda x{:}\, t.\, M : t \to t' \Rightarrow \lambda x{:}\, \langle\!\langle t \rangle\!\rangle.\, a} \qquad \frac{A \vdash M : t' \to t \Rightarrow a \qquad A \vdash e' : t' \Rightarrow a'}{A \vdash M\ e' : t \Rightarrow a\ a'}$$

$$\frac{A \vdash M : t \Rightarrow a \qquad \alpha \notin FV(A)}{A \vdash \Lambda\alpha.\, M : \forall\alpha.t \Rightarrow [a : \forall\alpha.\langle\!\langle t \rangle\!\rangle]} \qquad \frac{A \vdash M : \forall\alpha.t' \Rightarrow a}{A \vdash M\ t : t'\{t/\alpha\} \Rightarrow \langle a \rangle}$$

Since the translation rules copy the typing rules of system F, the translation is defined for all well-typed terms. There is no ambiguity and the translation is deterministic.

Lemma 34 *For any term M of system F, if $A \vdash M : t \Rightarrow a$, then $\langle\!\langle A \rangle\!\rangle \vdash a : \langle\!\langle t \rangle\!\rangle$.*

Proof: The proof is by structural induction on M. The only difficulty is to ensure that when typing $\langle a \rangle$ the polytype $[\sigma]^\epsilon$ of a is always anonymous. This is immediate: since the translation of all abstractions is annotated with the exact type of the variable, the unique label may always be quantified in the environment; therefore there are no free labels in the environment, and rule ELIM will always succeed. ■

If we choose for system F the semantics where type abstraction does not stop evaluation (*i.e.* $\Lambda\alpha.\, E$ is an evaluation context whenever E is), then the translation preserves the semantics in a strong sense (reduction steps of a term can be mapped to the reduction of the translated term). Another semantics would need easy adjustment, either of the translation or of the semantics of our system.

Let us compare a term M of system F with its translation a in our language, syntactically. Our types differ by having an extra type constructor $[\_]$ surrounding any polymorphic type. Our term variables do not carry type information. Lambda abstractions carry exactly the same type information in both M and a. The type information at elimination of polymorphism is always omitted in a. The counterpart is that type information at introduction of polymorphism appears explicitly in $[a : \forall\alpha.\langle\!\langle \tau \rangle\!\rangle]$. In $\Lambda\alpha.\, M$, only variable α is mentioned; the type τ is deduced from the type information located at application nodes in M.

The difference can be illustrated on the following example:

$$\langle\!\langle \lambda f{:}\, t_f.\, \lambda x{:}\, t_x.\, (f\ \tau_{fx})\ x \rangle\!\rangle = \lambda f{:}\, \tau_f.\, \lambda x{:}\, \tau_x.\, \langle f \rangle\ [x : \tau_x]$$

Type expressions with similar indices correspond to one another. The type $\tau_f x$ is such that the type $\tau_f\ \tau_{fx}$ reduces to an arrow of domain τ_x. The difference between the two approaches reduces to putting the type annotation t_{fx} on the function or the annotation τ_x on the argument. It is difficult to tell which option is more user-friendly. Obviously, examples can be found to make either side shorter. On the one hand, it could be argued that in many cases $\tau_f x$ is likely to be a subterm of τ_x, which favors system F. For instance, when polymorphic map is applied to a list of integers, t_{fx} is *int* and τ_x is *list@int*. On the other hand, our language is also more flexible: type annotations

are mandatory in system F, but not in our proposal. In particular, ML programs do not require any explicit type information at all. That is, in the above example, *list int* would not need to be provided since it would be fully inferred. While functions are often polymorphic, their arguments are frequently monomorphic.

There is another insignificant, but interesting difference betweem the two approaches. Ours allows for multiple abstractions to be introduced simultaneously, as in $[a : \forall\alpha_1, \alpha_2.\tau]$. Since type application is explicit in system F, the expression $\Lambda\alpha_1, \alpha_2.\, M$ would be ambiguous; thus it is not allowed. This does not give us more concision that system F, but it allows to avoid the common pattern $[[f : \forall\alpha.\tau] : \forall\alpha'.\forall\alpha.\tau]$. In most cases, instantiation of all variables will be simultaneous and we can simply write $[f : \forall\alpha'.\forall\alpha.\tau]$.

The simplicity of our encoding of system F compared to its encoding into polymorphic datatypes is permitted by the introduction of polytypes as first-class types, and does not rely on the inference of polytypes at their elimination points. As we have shown by using only one label in the translation, if we made the elimination of polymorphism always explicit, we could keep first-class polytypes and omit all labels in polytypes. We would obtain a weaker but simpler proposal that would still extend ML and be as powerful as system F, however more verbose.

7.4 Application to Objective ML

In this section we show how the core language can be used to provide polymorphic methods in Objective ML[2] [113]. Polymorphic methods are useful in parameterized classes. Indirectly, they may also reduce the need for explicit coercions.

While Objective ML has parametric classes, it does not allow methods to be polymorphic. For instance, the following class definition fails to type.

```
let α collection = class (l)
    val contents = l
    meth mem = λx. mem x contents
    meth fold : (β → α → β) → β → β
              = λf.λx. fold_left f x contents
end
```

The reason is that variable β is free in the type for method `fold` and it is not bound to a class parameter. The solution is to have the method `fold` be polymorphic in β. With polytypes, we can write

```
meth fold = [λf.λx. fold_left f x contents
            : ∀β. (β → α → β) → β → β]
```

Still, we have to distinguish between polymorphic and monomorphic methods, in particular when sending a message to the object. The aim of the remainder of this section is to make invocation polymorphic and monomorphic methods similar, and more generally, to make the invocation of polymorphic methods lighter.

The first step is to give polytypes to all methods. This is easily done by wrapping monomorphic methods into polytypes. For instance, we shall write

```
meth mem = [λx. mem x l : α]
```

[2]The examples of objects and classes given below are rather intuitive, and could be translated in other class-based object-oriented languages; the reader may refer to [113] for a formal presentation of Objective ML.

ELIM
let $\alpha \notin FV(A)$ **in**
if $A \triangleright a : \alpha \Rrightarrow \exists \bar{\xi}.\, \theta$ **then**
if $\theta(\alpha) = [\forall \bar{\alpha}'.\tau']^{\epsilon}$ **and** $\epsilon \notin FL(\theta(A))$
then $A \triangleright \langle a \rangle : \tau \Rrightarrow \exists \bar{\xi}, \alpha, \bar{\alpha}'.\, \theta \wedge \tau' = \tau$
else if $\theta(\alpha) = \alpha'$ **and** $\alpha' \notin FV(\theta(A))$ **then** $A \triangleright \langle a \rangle : \tau \Rrightarrow \exists \bar{\xi}, \alpha.\, \theta$
else let $\epsilon' \notin (FL(A) \cup FL(\tau))$ **in** $A \triangleright \langle a \rangle : \tau \Rrightarrow \exists \bar{\xi}, \epsilon', \alpha.\, \theta \wedge \alpha \doteq [\tau]^{\epsilon'}$
else $A \triangleright \langle a \rangle : \tau \Rrightarrow \bot$

Figure 7.5: Type inference rule for use of monomorphic polytypes

However, we still want to be able to use monomorphic methods without type annotations. There is a small but very convenient extension to the core language that solves this problem. We add a new typing rule ELIM-M:

$$\frac{A \vdash a : [\tau]^{\epsilon}}{A \vdash \langle a \rangle : \tau} \text{(ELIM-M)}$$

As opposed to rule ELIM, this one allows ϵ to appear in A. Inference problems are solved by forcing the polytype to be monomorphic.

Both rules ELIM and ELIM-M apply when ϵ is anonymous and the polytype is monomorphic, but they produce the same derivation. If either ϵ is free in A, or the polytype is polymorphic, then only one of the two rules may be used. As a result, principal types are preserved. The type inference algorithm can be modified as shown in figure 7.5. The subject reduction property is preserved.

The expression $(\lambda x.\, \lambda y.\, \langle x \# \texttt{mem} \rangle\ y)$ is then typable with principal type $\langle \texttt{mem} : [\alpha \to \beta]; .. \rangle \to \alpha \to \beta$. Since all methods are now given polytypes, we shall change our notations (the new notations are given in term of the old ones): in types, we now write $m : \sigma$ for $m : [\sigma]$; in expressions, we now write $m : \sigma = a$ for $m = [a : \sigma]$, $m = a$ for $m = [a : \alpha]$ and $a \# m$ for $\langle a \# m \rangle$. With the new notations, the collection example is written:

```
let α collection = class (l)
    val contents = l
    meth mem = λx. mem x contents
    meth fold : ∀β. (β → α → β) → β → β
              = λf.λx. fold_left f x contents
end;;
value collection : class α (α list)
    meth mem : α → bool
    meth fold : ∀β. (β → α → β) → β → β
end
```

A monomorphic method is used exactly as before.

```
let coll_mem c x = c#mem x
coll_mem : ⟨mem : α → β; ..⟩ → α → β
```

However, when polymorphic methods are used under abstractions, the type of the object should be provided as an annotation,

```
let simple_and_double (c : α collection) =
    let l1 = c#fold (λx.λy. x::y) [] in
    let l2 = c#fold (λx.λy. (x,x)::y) [] in
    (l1, l2);;
simple_and_double : α collection → (α list * (α * α) list)
```

Since the method `fold` is used with two different types, this example could not be typed without first-class polymorphism.

Polymorphic methods also appear to be useful to limit the need for explicit coercions. In Objective ML, coercions are explicit. For instance, assume that objects of class point have the interface $\langle$`x : int; y : int`$\rangle$, and that we want to define a class circle with a method giving the distance from the circle to a point.

```
let circle = class (x,y,r)  ...
    meth distance = λp:point. ...
end;;
value circle : class (int * int * int)  ...
    meth distance : point → float
end
```

Given a point `p` and a circle `c`, we compute their distance by `c#distance p`. However, an object `cp` of a class `color_point` where `color_point` is a subtype of `point` (*e.g.* its interface is $\langle$`x : int; y : int; color : color`$\rangle$) needs to be explicitly coerced to `point` before its distance to the circle can be computed:

```
c#distance (cp : color_point :> point)
```

This coercion could be avoided if `distance` were a toplevel function rather than a method:

```
let distance c p = c#distance (p :> point);;
value distance : ⟨distance : point → α ; ..⟩ → #point → α
```

The type expression `#point` represents any subtype of `point`. Actually, it is an abbreviation for the type $\langle$`x : int; y : int;` $\rho\rangle$. Here, `#point` contains a hidden row variable that is polymorphic in the function `distance`. This allows different applications to use different instances of the generic row variable and thus to accept different objects all matching the type of points.

Explicit polymorphism allows to recover the same power inside methods:

```
meth distance : ∀α:#point. α → float = λp. ...
```

Then, `c#distance cp` is typable just by instantiation of these row variables, without explicit coercion. Of course, we must know here that `c` is a circle before using method `distance`, like would happen in more classical object-oriented type systems. There is an alternative between using explicit coercions or providing more type information. The advantage of type information is that it occurs at more convenient places. That is, it is necessary in method definitions and at the invocation of a method of an object of unknown type. On the opposite, explicit coercions must be repeated at each invocation of a method even when all types are known.

7.5 Value-only polymorphism

For impure functional programming languages, value-only polymorphism has become the standard way to handle the ubiquity of side-effects. It preserves type-soundness in the presence of side-effect, without making the type system overly complex. It is based on a very simple idea —if an

expression is *expansive*, *i.e.* its evaluation may produce side-effects, then its type should not be polymorphic [124].

This is usually incorporated by restricting the GEN rule to a class of expressions b, called non-expansive, composed of variables and functions. Equivalently, this restriction can be put on the LET rule: both ways give exactly the same canonical derivations in the core language. We actually prefer the latter, since we also need rule GEN to precede rules ELIM and Intro.

Thus, we replace rules INTRO and LET by the following four rules, each rule being split in its expansive and non-expansive versions.

$$\text{(POLY-V)}\ \frac{A \vdash b : \sigma_1 \qquad (\sigma_1 : \sigma : \sigma_2)}{A \vdash [b : \sigma] : [\sigma_2]^\epsilon} \qquad \text{(POLY-E)}\ \frac{A \vdash a : \tau_1 \qquad (\tau_1 : \tau : \tau_2)}{A \vdash [a : \tau] : [\tau_2]^\epsilon} \qquad \text{(LET-V)}\ \frac{A \vdash b : \varsigma \qquad A \oplus x : \varsigma \vdash a : \tau}{A \vdash \mathtt{let}\ x = b\ \mathtt{in}\ a : \tau}$$

$$\text{(LET-E)}\ \frac{A \vdash a_1 : \tau' \qquad A \oplus x : \tau' \vdash a_2 : \tau}{A \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau}$$

The class of non-expansive expressions can be refined, provided the evaluation cannot produce side-effects and preserves non-expansiveness. For instance, in ML, we can consider let-bindings of non-expansive expressions in non-expansive expressions as non-expansive. In our calculus, type annotations are also non-expansive. More generally, any expression where every application is protected (*i.e.* appears) under an abstraction is non-expansive (creation of mutable data-structure would be the application of a primitive):

$$b ::= x \mid \lambda x.\, a \mid \mathtt{let}\ x = b\ \mathtt{in}\ b \mid (b : \tau) \mid [b : \sigma] \mid \langle b \rangle$$

This system works perfectly, and all properties are preserved.

However, it seems too weak in practice. Since we use polymorphism of ϵ's to denote confirmation of polytypes, as soon as we let-bind an expansive expression, all its ϵ's become monomorphic, and all its polytypes need an explicit type annotation before they can be eliminated. For instance, the following program is not typable, because labels in the type of the binding occurrence of g cannot be generalized.

$$\mathtt{let}\ f = [\lambda x.\, x : \forall\alpha.\alpha \to \alpha]\ \mathtt{in}\ \mathtt{let}\ g = (\lambda x.\, x)\ f\ \mathtt{in}\ \langle g \rangle\ g$$

When ML polymorphism is restricted to values, the result of an application is monomorphic (here, the result of applying $\lambda x.\, x$ to f). Traditionally, the typical situation when a polymorphic result is restricted to be monomorphic is partial application. There, polymorphism is easily recoverable by η-expansion. However, the same problem appears when objects are represented as records of methods, with no possibility of η-expansion. In our core language, the only way to recover at least explicit polymorphism in such a case is to annotate the use of let-bound variables with their own types:

$$\mathtt{let}\ f = [\lambda x.\, x : \forall\alpha.\alpha \to \alpha]\ \mathtt{in}\ \mathtt{let}\ g = (\lambda x.\, x)\ f\ \mathtt{in}\ \langle g : [\forall\alpha.\alpha \to \alpha] \rangle\ g$$

In practice, with objects, this means recalling explicit polymorphism information at each method invocation. The strength of our system being its ability to omit such information, its interest would be significantly reduced by this limitation.

One might think that allowing quantification on ϵ in LET-E, *i.e.* write $\forall\bar{\epsilon}.\tau'$ in place of τ', is harmless. Indeed, ϵ's polymorphism does not allow type mismatches like α's polymorphism would:

verifying identity of type schemes is done separately. However, this rule would break principal types. Consider, for instance, the following expression:

$$\texttt{let}\ x = \texttt{id}\ []\ \texttt{in}\ \texttt{let}\ y = \langle \texttt{hd}\ x \rangle\ \texttt{in}\ x$$

It can be assigned type $[\sigma]^\epsilon$ `list` for any type scheme σ. Since type schemes of polytypes are not ordered, there is no principal type for this expression.

This problem is pathological, but not anecdotical. It can be solved by restricting to minimal judgments. That is, we replace LET-V and LET-E by the following restricted rules. $A \vdash^\star a : \varsigma$ means that ς is a minimal type scheme for a under assumptions A, *i.e.* there exists no ς' strictly greater than ς in the instantiation order, such that $A \vdash a : \varsigma'$. (Since we happen to be keeping principality, ς is the principal scheme for a under assumptions A.)

$$\text{(LET-V}^\star\text{)}\quad \frac{A \vdash^\star b : \varsigma \qquad A \oplus x : \varsigma \vdash a : \tau}{A \vdash \texttt{let}\ x = b\ \texttt{in}\ a : \tau} \qquad \text{(LET-E}^\star\text{)}\quad \frac{A \vdash^\star a_1 : \forall \bar\epsilon \bar\alpha.\tau' \qquad a_1 \not\equiv b \qquad A \oplus x : (\forall \bar\epsilon.\tau')\{\bar\tau/\bar\alpha\} \vdash a_2 : \tau}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

The rule LET-E* may seem strange, since it is not an instance of the original LET rule, but rather a combination of INST and LET. The original derivation would have been:

$$\frac{\dfrac{\dfrac{A \vdash a_1 : \forall \bar\epsilon \bar\alpha.\tau'}{\vdots}}{A \vdash a_1 : \forall \bar\epsilon.\tau''} \qquad A \oplus x : \forall \bar\epsilon.\tau'' \vdash a_2 : \tau}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

The restriction to principal judgments is not new: it has already been used for the typing of dynamics in ML [72], for instance. One has to reject the program $\lambda x.(\texttt{dynamic}\ x)$ because, in the principal judgment $x : \alpha \vdash x : \alpha$, some variable of the type of x occurs free in the context. A non principal judgment obtained by choosing `int` for α would be correct, but arbitrary. More recently, it has been used for local type inference in system $F_\leq$ [96]. Type inference is only allowed locally at application nodes, and upon the condition there is a principal solution to the local inference problem. Without this condition, choices made at an application node would influence other nodes, and inference would loose its locality.

We use minimality here in a somewhat different way. In the above two systems, requiring a principal solution was a way to have the inference fail on some ambiguous cases. Contrary to dynamics, our types do not need to be ground; they may share variables with the environment. Contrary to local type inference, all our satisfiable inference problems have principal solutions. Thus, our minimality condition never makes a type inference problem fail, but only restricts the set of types that can be assigned to a variable in a let statement. Notice that $\vdash^\star$ judgments do not actually require the derivation to be principal, but only minimal; they do not eliminate all different derivations, but only those that would be obtained by unnecessarily instantiating some types. We may then prove the existence of principal types by showing that all minimal schemes are equal modulo renaming of bound variables, and as a result our minimality condition happens to be a principality condition. This condition is not harmful when reasoning about derivations: the property of minimality of a derivation is kept by substitution of free type variables, so that the stability lemma is still valid in the extended system.

Still, we do not consider this solution as fully satisfactory, and we view it as an example of the difficulties inherent to value-only polymorphism.

7.6 Related Work

Full type inference of polymorphic types is undecidable [123]. Several works have studied the problem of partial type inference in system F.

Some implementations of languages based on system F relieve the user from the burden of writing all types down. In Cardelli's implementation of the language Fun [23] polymorphic types are marked either as implicit (actually their variables are marked) and they are automatically instantiated when used, or as explicit and they remain polymorphic until they are explicitly instantiated This mechanism turns out to be quite effective in inferring type applications. However, types of abstracted values are never inferred. Thus, the expression $\lambda x.\, x$ cannot be typed without providing a type annotation on the variable x, which shows that this is not an extension to ML. Pierce and Turner have extended this partial inference mechanism to $F^{\omega}_{\leq}$ in the design of the language Pict [95]. By default they also assign "unification variables" to parameters of functions with no type annotations. Their solution requires surprisingly little type information in practice, especially in the absence of subtyping. Still, as for Cardelli's solution, it is quite difficult to know exactly the set of well-typed programs, since the description is only algorithmic.

Conscious of this problem, they more recently proposed to replace this unpredictable approach by one based on predictable local inference [96, 94]. Their approach is somewhat opposite of ours: while we provide some inference-free type checking without modifying ML type inference, they add some type inference to $F_{\leq}$ and keep a checking based system. In their approach, the uniqueness of typing is still valid at every step. As we, they distinguish between the specification and the algorithm of type inference, but this distinction is only limited to one rule, the one doing local inference. This rule has two provably equivalent versions: one is a specification of the inferred type in terms of a universal property; the other one is algorithmic and is presented in a constraint-solving style. The difference of approach and the fact that they also handle subtyping make it difficult to compare the respective strength of the two systems.

A different approach is taken by Pfenning [89]. Instead of providing type annotations on lambda's, he indicates possible type applications (this corresponds to the notation $\langle \_ \rangle$ in our language). Then, he shows that partial type inference in system F corresponds to second-order unification and is thus undecidable [90]. As ours, his solution is an extension of ML. It is also more powerful; the price is the loss of principal types and decidability of type inference. However, a decidable subcase of higher-order unification has also been considered in [37]. Neither solution handles subtyping yet.

Kfoury and Wells show that type inference could be done for the rank-2 fragment of system F [59]. However, they do not have a notion of principal types. It is also unclear how partial type information could be added.

In [83], Läufer and Odersky actually present two different mechanisms. First, as we explained in the introduction they add higher-order polymorphism with fully explicit introduction and elimination. As we have seen, our framework subsumes theirs. They also introduce another mechanism that allows annotations of abstractions by type schemes as in $\lambda x{:}\sigma.\, x$ together with a type containment relation on type schemes similar to the one of Mitchell [80] but with some serious restriction. Type schemes may be of the form $\forall\alpha.\sigma_1 \to \sigma_2$, where σ_i are type scheme themselves. However, universal variables such as α can only be substituted by simple types. Thus, the only way to apply a function of type $\forall\alpha.\alpha \to \alpha$ to a polymorphic value remains to embed the argument inside an explicitly defined polytype. Actually, one of the reasons for complementing universal-datatype polymorphism by restricted type-containment is to obtain an encoding of system F. In our case, the encoding of system F is permitted by the use of polytypes.

In [39], Duggan proposes an extension to ML with objects and polymorphic methods. His solution heavily relies on the use of kinds and type annotations. These are carried by method names that must be declared before being used. In this regard, his solution is similar to having fully explicit polymorphism both at introduction and elimination, as in [83]. His use of recursive kinds allows some programs that cannot be typed in our proposal (section 7.4). However, this is due to a different interpretation of object types rather than a stronger treatment of polymorphism.

Conclusion

We have presented a conservative extension to ML that allows for first-class polytypes and first-class polymorphic values. In our proposal as in ML, let-polymorphism remains implicit. While first-class polymorphism must be introduced explicitly, type information is inferred at the elimination point. This allows for polymorphic methods in Objective ML, which are particularly useful in parametric classes.

We have also shown that polymorphism can be restricted to values, so as to be sound in the presence of side-effects. This naive standard restriction weakens the propagation of first-class polymorphism, and forces unnecessarily some type annotations. Thus, we have also proposed an extension that covers all useful cases and does not present any known limitations. Even though the specification of typechecking becomes technically more difficult, since it involves the notion of minimal judgements, the principal-type property is preserved. Although practically insignificant, this difficulty exposes a drawback of the value-only restriction of polymorphism.

As future work, two extensions of importance are to be studied. Firstly, we should consider applying our technique to existential types. The encoding of these into universal types introduces inner quantifiers, which removes all opportunities for inference. It remains unclear whether primitive existential types could benefit from our work. Secondly, the replacement of the core ML type system by one with subtyping constraints as in [4, 40], would combine first-order generic polymorphism and subtyping polymorphism in an ML-like language. The issues of constraint checking and type generalization are rather orthogonal. However, some recent and more general presentation [97, 41] significantly differs from ML. Thus, more investigation is required.

The principle of our approach was to keep type inference first-order. While we believe this to be sufficient in practice, we would still like to formulate our type system in terms of partial type inference for second-order lambda-calculus.

Appendix

7.7 Proofs of main theorems

Lemmas 32 (*canonical derivations*) and 33 (*stability by substitution*) are tedious but essential in ML. Their proofs easily cary over with the three new rules, Ann, Intro, and Elim.

Proof of type soundness for the core language

Lemma 35 (Term substitution) *If* $A \oplus x : \sigma_2 \vdash a : \sigma_1$ *and* $A \vdash v : \sigma_2$ *hold, then* $A \vdash a\{v/x\} : \sigma_1$ *also holds.*

Proof: The proof is an easy induction on the structure of v. ■

Theorem 1 (Subject reduction) *Reduction preserves typings,* i.e. *if* $a_1 \longrightarrow a_2$, *then* $a_1 \subset a_2$.

Proof: We show that every rule in the definition of $\longrightarrow$ is satisfied by the relation $\subset$. Since $\longrightarrow$ is the smallest relation verifying those rules, then $\subset$ must be a super-relation of $\longrightarrow$. All cases are independent. In each case but CONTEXT, we assume that $A \vdash a_1 : \varsigma$ (1) and that $a_1 \longrightarrow a_2$, (the structure of a_1 depending on the case) and we show that $A \vdash a_2 : \varsigma$ (2).

We first assume that the derivation does not end with a rule GEN. If the derivation ends with a rule GEN, it is of the form:

$$\dfrac{\dfrac{\Delta}{A \vdash a_1 : \varsigma_0}}{A \vdash a_1 : \forall \bar{\xi}.\varsigma_0}\ (\text{GEN}^*)$$

where the derivation Δ of (1) does not end with a rule GEN. Thus we have $A \vdash a_2 : \varsigma_0$ and (2) follows by the same sequence of generalizations.

Case FUN and LET: This is a straightforward application of term-substitution lemma.

Case ELIM: A canonical derivation of (1) ends with

$$\dfrac{\dfrac{\dfrac{A \vdash a : \sigma_1 \qquad (\sigma_1 : \sigma_0 : \sigma_2)}{A \vdash [a : \sigma_0] : [\sigma_2]^\epsilon}\ (\text{INTRO})}{A \vdash [a : \sigma_0] : \forall \epsilon.[\sigma_2]^\epsilon}\ (\text{GEN})}{A \vdash \langle [a : \sigma_0] \rangle : \sigma_2}\ (\text{ELIM})$$

The type schemes σ_1, σ_0, and σ_2 are of the form $\forall \bar{\alpha}.\tau_1$, $\forall \bar{\alpha}.\tau_0$, and $\forall \bar{\alpha}.\tau_2$, and such that $(\tau_1 : \tau_0 : \tau_2)$. Choosing variables $\bar{\alpha}$ that do not occur free in A, we can contract this derivation into the following derivation of (2):

$$\dfrac{\dfrac{\dfrac{A \vdash a : \sigma_1}{A \vdash a : \tau_1}\ (\text{INST}^*) \qquad (\tau_1 : \tau_0 : \tau_2)}{A \vdash (a : \tau_0) : \tau_2}\ (\text{ANN})}{A \vdash (a : \tau_0) : \sigma_2}\ (\text{GEN}^*)$$

Case TFUN: A canonical derivation of $A \vdash a_1 : \sigma$ ends with

$$\dfrac{\dfrac{A \vdash v_1 : \tau_2' \to \tau_1' \qquad (\tau_2' \to \tau_1' : \tau_2 \to \tau_1 : \tau_2'' \to \tau_1'')\ (3)}{A \vdash (v_1 : \tau_2 \to \tau_1) : \tau_2'' \to \tau_1''}\ (\text{ANN}) \qquad A \vdash v_2 : \tau_2''}{A \vdash (v_1 : \tau_2 \to \tau_1)\ v_2 : \tau_1''}\ (\text{APP})$$

Since the relation (3) implies both $(\tau_2'' : \tau_2 : \tau_2')$ and $(\tau_1' : \tau_1 : \tau_1'')$, we can build the derivation:

$$\dfrac{\dfrac{A \vdash v_1 : \tau_2' \to \tau_1' \qquad \dfrac{A \vdash v_2 : \tau_2'' \qquad (\tau_2'' : \tau_2 : \tau_2')}{A \vdash (v_2 : \tau_2) : \tau_2'}\ (\text{ANN})}{A \vdash v_1\ (v_2 : \tau_2) : \tau_1'}\ (\text{APP}) \qquad (\tau_1' : \tau_1 : \tau_1'')}{A \vdash (v_1\ (v_2 : \tau_1) : \tau_2) : \tau_1''}\ (\text{ANN})$$

Case TINT**:** The last derivation of (1) ends with:

$$\dfrac{\dfrac{A \vdash v : \sigma_1' \ (3) \qquad (\sigma_1' : \sigma_1 : \sigma_1'') \ (4)}{A \vdash [v : \sigma_1] : [\sigma_1'']^{\epsilon_1}} \ (\textsc{Intro}) \qquad ([\sigma_1'']^{\epsilon_1} : [\sigma_2]^{\epsilon_2} : [\sigma_3]^{\epsilon_3}) \ (5)}{A \vdash ([v : \sigma_1] : [\sigma_2]^{\epsilon_2}) : [\sigma_3]^{\epsilon_3}} \ (\textsc{Ann})$$

Let $\forall \bar{\alpha}.\tau_1$ be σ_1. From (4), we know that we can write σ_1' and σ_1'' as $\forall \bar{\alpha}.\tau_1'$ and $\forall \bar{\alpha}.\tau_1''$. Moreover, we have $(\tau_1' : \tau_1 : \tau_1'')$. From (5), we also get $(\sigma_1'' : \sigma_2 : \sigma_3)$. Thus, we have

$$\dfrac{\dfrac{\dfrac{(3)}{A \vdash v : \tau_1'} \ (\textsc{Inst}^*) \qquad (\tau_1' : \tau_1 : \tau_1'')}{A \vdash (v : \tau_1) : \tau_1''} \ (\textsc{Ann}) \quad}{A \vdash (v : \tau_1) : \sigma_1''} \ (\textsc{Gen}^*) \qquad (\sigma_1'' : \sigma_2 : \sigma_3)}{A \vdash [(v : \tau_1) : \sigma_2] : [\sigma_3]^{\epsilon_3}} \ (\textsc{Intro})$$

Case TVAR**:** Annotating with a type variable does nothing.

Case CONTEXT**:** Here, we need to show that if $a_1 \subset a_2$ then for any evaluation context E we also have $E\{a_1\} \subset E\{a_2\}$. The proof is by structural induction on E. All cases are immediate since evaluation contexts do not contain any binding.

■

Theorem 2 (Canonical forms) *Irreducible programs that are well-typed in the empty environment are values.*

Proof: We first relate the shape of types and the shape of values. Let v be a value of type τ. By considering all possible canonical derivations, we see that:

- if v is a poly expression, possibly with a type constraint, then τ is a polytype;
- otherwise, v is of the form w and τ is a functional type.

Since polytypes and functional types are incompatible, we can invert the property:

- if τ is a polytype, then v is a poly expression, possibly with a typed constraint.
- otherwise, τ is a functional type, and v is of the form w.

Then, the theorem follows: considering a program a that is well-typed in the empty environment and that cannot be reduced, it can easily be shown by structural induction that a is a value. ■

Proof of the principal type property

Lemma 36 (unification) *Each of the rules given in figures 7.2 and 7.3, is correct and complete.*

Proof:

Cases OCCUR-CHECK**,** MERGE**,** ABSORB**, and** DECOMPOSE**:** those are standard rules for first-order unification.

Case DECOMPOSE-POLY **and** CLASH**:** immediate.

Case POLYTYPES**:** This case amounts to fully formalizing the discussion in section 7.2.4. Assume that $\bar{\alpha} \cap \bar{\alpha}'$, $\bar{\alpha} \cap FV(\tau')$, and $\bar{\alpha}' \cap FV(\tau)$ are all empty (1).

Soundness: Assume that θ is a solution of $\exists\, \bar{\alpha}\bar{\alpha}'.\, \tau \doteq \tau' \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}'$. Let η be a renaming of $\bar{\alpha}\bar{\alpha}'$ into variables outside of free variables of θ, τ, τ', and $\bar{\alpha}\bar{\alpha}'$. The substitution $\eta \circ \theta$ is also a solution of the same unificand. Since its image has no variable in common with $\bar{\alpha}\bar{\alpha}'$, the substitution $\eta \circ \theta \setminus \bar{\alpha}\bar{\alpha}'$ can be extended by a substitution ρ of domain $\bar{\alpha}\bar{\alpha}'$ such that the substitution θ' equal to $\eta \circ \theta \setminus \bar{\alpha}\bar{\alpha}'$ is a solution of $\tau \doteq \tau' \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}'$. Since θ' is a solution of $\tau \leftrightarrow \tau'$, the substitution ρ is injective on $\bar{\alpha}$ and $\bar{\alpha}'$ taken separately. Moreover, its image is in $\bar{\alpha}\bar{\alpha}'$. The subsitution $(\eta + \eta^{-1}) \circ \theta'$ decomposes as $(\theta \setminus \bar{\alpha}\bar{\alpha}') + (\eta \circ \rho)$, which is actually equal to $\theta \circ \eta \circ \rho$; it must be a solution of $\tau \doteq \tau'$. Therefore the subsitution θ is a solution of $\forall\bar{\alpha}.\tau \doteq \forall\bar{\alpha}'.\tau'$.

Completeness: Let θ be a solution of $\forall\bar{\alpha}.\tau \doteq \forall\bar{\alpha}'.\tau'$. Reusing the reasoning and the definitions of section 7.2.4, the substitution $(\eta \circ \theta \setminus \bar{\alpha}\bar{\alpha}') + \rho$ is a solution of $\tau \doteq \tau' \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}'$ where η is a renaming of $\bar{\alpha}\bar{\alpha}'$ into variables taken outside of free variables of θ, τ, τ', and $\bar{\alpha}\bar{\alpha}'$. Thus, $\eta \circ \theta$ is a solution of $\exists\, \bar{\alpha}\bar{\alpha}'.\, \tau \doteq \tau' \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}'$ and so is θ by composition with η^{-1}.

Case RENAMING-TRUE**:** The completeness is obvious. For the soundness, let θ be any substitution. Let η be a renaming of $\bar{\alpha}\bar{\alpha}'$ outside of $\bar{\alpha}\bar{\alpha}'$ and free variables of θ. The substitution $(\eta \circ \theta) \setminus \bar{\alpha}\bar{\alpha}'$ can be extended with the substitution $(\alpha_i \mapsto \alpha_i')^{i \in 1..n}$. Clearly, this extension satisfies both $(\alpha_i \doteq \alpha_i')^{i \in 1..n}$ and $\bar{\alpha} \leftrightarrow \bar{\alpha}'$. Thus θ is a solution of $\exists\, \bar{\alpha}\bar{\alpha}'.\, (\alpha_i \doteq \alpha_i')^{i \in 1..n} \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}'$.

Case RENAMING-FALSE**:** The soundness is obvious. For the completeness let us consider the two following cases:

$\beta \in \bar{\alpha}$ **and** $\tau \notin \bar{\alpha}' \cup \{\beta\}$: Assume that there exists a solution θ of both $\beta \doteq \tau \doteq e$ and $\bar{\alpha} \leftrightarrow \bar{\alpha}'$. Since $\theta(\tau)$ is equal to $\theta(\alpha)$, it must be a variable, and so should τ itself. Since $\theta \setminus \bar{\alpha}\bar{\alpha}'$ should not have variables in common with $\bar{\alpha}\bar{\alpha}'$, τ must be in $\bar{\alpha}\bar{\alpha}'$. However, since it is not in $\bar{\alpha}'$, it must be another variable γ of α distinct from β, which contradicts with the fact that $\theta\restriction\bar{\alpha}$ must be injective (condition 1).

$\beta \in \bar{\alpha} \cap FV(\tau)$ **and** $\tau \neq \beta$: In particular, τ must be a proper term. Assume that there exists a solution θ of both $\gamma \doteq \tau \doteq e$ and $\bar{\alpha} \leftrightarrow \bar{\alpha}'$. The term $\theta(\gamma)$, equal to $\theta(\tau)$, is a proper term; thus, γ cannot be a variable of $\bar{\alpha}\bar{\alpha}'$. However, $\theta(\gamma)$ contains the variable $\theta(\beta)$ that belongs to $\bar{\alpha}\bar{\alpha}'$. This contradicts condition 3.

■

Theorem 4 *Given a typing problem $(A \triangleright a : \tau)$ there exists a principal solution, which is computed by the set of rules described in figures 7.2, 7.3 and 7.4, or there is no solution and the problem reduces to $\bot$.*

Proof: We first show the soundness and completeness of each rewriting rule:

Cases VAR, FUN, APP, **and** LET**:** are as in ML.

Case ANN**:** The case ANN is not special since the construct $(\_ : \tau)$ could be treated as the application of a primitive.

Case INTRO**:** We assume that all the conditions of the first four lines are satisfied. We write σ_i for $\sigma\{\bar{\alpha}_1\bar{\epsilon}_i/\bar{\alpha}_0\bar{\epsilon}_0\}$.

Soundness: Let us assume that $A \vdash a : \tau_1\{\bar{\epsilon}_1/\bar{\epsilon}_0\} \Rrightarrow \exists\,\bar{\xi}.\,\theta$ and $\bar{\alpha} \cap dom(\theta) \cup FV(im(\theta)) = \emptyset$. We have $\theta(A) \vdash a : \theta(\sigma_1)$ by generalization of $\bar{\alpha}$ in the judgment $\theta(A) \vdash a : \theta(\tau_1\{\bar{\epsilon}_1/\bar{\epsilon}_0\})$. Since by construction $(\theta(\sigma_1) : \sigma : \theta(\sigma_2))$, we also have $\theta(A) \vdash [a : \sigma] : \theta([\sigma_2]^\epsilon)$. That is, θ is a solution of $A \vdash [a : \sigma] : [\sigma_2]^\epsilon$. Thus, a solution of $\theta \wedge \tau = [\sigma_2]^\epsilon$ is a solution of $A \vdash [a : \sigma] : \tau$. Moreover, no variable of $\bar{\epsilon}_1, \bar{\epsilon}_2, \epsilon, \bar{\alpha}_1$ appears in A or τ.

Completeness: Let us assume that θ' is a solution of $A \triangleright [a : \sigma] : \tau$. A canonical derivation of $\theta'(A) \vdash [a : \sigma] : \theta'(\tau)$ must end with rule INTRO. Thus, there exists some type schemes σ'_1 and σ'_2 and some label ϵ such that $\theta'(A) \vdash a : \sigma'_1$ (1), $(\sigma'_1 : \sigma : \sigma'_2)$ (2), and $\theta'(\tau) = [\sigma]^\epsilon_2$ (3). By definition of the relation $(\_ : \sigma : \_)$ the pair (σ'_1, σ'_2) must be of the form $(\theta''(\sigma_1), \theta''(\sigma_2))$ for some substitution θ'' of domain $\bar{\epsilon}_1\bar{\epsilon}_2\bar{\alpha}_0$. A canonical derivation of (1) must end with a succession of rules GEN. Thus we have $\theta'(A) \vdash a : \theta''(\tau_1\{\bar{\epsilon}_1/\bar{\epsilon}_0\})$. On the one hand, the substitution $\theta' + \theta''$ is a solution of $A \vdash a : \tau_1\{\bar{\epsilon}_1/\bar{\epsilon}_0\}$, and consequently a solution of θ. On the other hand, it is a solution of $\tau = [\tau_1\{\bar{\epsilon}_2/\bar{\epsilon}_0\}]^\epsilon$. Moreover, it extends θ' on $\bar{\alpha}_0$, $\bar{\epsilon}_0$, $\bar{\epsilon}_1$, ϵ, and $\bar{\xi}$.

The completeness of the else branch is straightforward; The proof above actually applies if θ is $\perp$. If θ is not $\perp$, the right condition may always be satisfied since $\bar{\alpha}$ is disjoint from free variables of the typing problem.

Case ELIM**:** We assume that the condition of the first line is satisfied.

Soundness: If $\theta(\alpha) = [\forall\bar{\alpha}'.\tau']^\epsilon$ and $\epsilon \notin FL(\theta(A))$ then rule ELIM applies, and an extension of θ such that $\theta(\tau) = \theta(\tau')$ is a solution of $A \triangleright \langle a \rangle : \tau$. If $\theta(\alpha) = \alpha'$ and $\alpha' \notin FV(\theta(A))$ then from $\theta(A) \vdash a : \alpha'$ we deduce $\theta(A) \vdash a : [\tau]^\epsilon$ for some ϵ not in $FV(\theta(A))$. By generalization of ϵ and rule ELIM, we get $\theta(A) \vdash \langle a \rangle : \theta(\tau)$. The substitution θ is thus a solution of $A \triangleright \langle a \rangle : \tau$.

Completeness: Let us assume that θ' is a solution of $A \vdash \langle a \rangle : \tau$. The canonical derivation of $\theta'(A) \vdash \langle a \rangle : \theta'(\tau)$ must end with rule ELIM. Thus, we must have $\theta'(A) \triangleright a : [\sigma]^{\epsilon'}$ for some ϵ' that does not appear in $\theta'(A)$ and some type scheme σ of which $\theta'(\tau)$ is an instance. Since $\exists\,\bar{\xi}.\,\theta$ is a principal solution of $A \triangleright a : \alpha$, θ' can be extended on $\bar{\xi}$ into a solution of $\theta \wedge \theta(\alpha) \doteq [\sigma]^{\epsilon'}$ (1).

Therefore $\theta(\alpha)$ cannot be an arrow type. If it is a variable, then it cannot belong to $\theta(A)$, otherwise ϵ' would belong to $\theta'(A)$. Hence, together with (1) the completeness of the second and third cases.

If $\theta(\alpha) = [\forall\bar{\alpha}'.\tau']^\epsilon$ then ϵ cannot belong to $FL(\theta(A))$, otherwise ϵ' would belong to $FL(\theta'(A))$. Since θ' is a solution of $[\sigma]^{\epsilon'} \doteq [\forall\bar{\alpha}'.\tau']^\epsilon$, it is also a solution of $\sigma = \forall\bar{\alpha}'.\tau$. Since $\theta'(\tau)$ is an instance of σ, it is an instance of $\forall\bar{\alpha}'.\tau$. Thus θ' can be extended on $\bar{\alpha}'$ into a solution of $\tau = \tau'$. Together with (1), θ' is a solution of $\theta \wedge \tau = \tau'$.

Termination: We now show that applying the rules in any order always terminates, with a unification problem in solved form.

Each rule of the algorithm decreases of the lexicographic ordering composed of successively

1. the sum of sizes of program components,
2. the sum of monomials $X^{size(\sigma)}$ for all type and type-scheme components of the system,
3. the number of polymorphic constraints,
4. the number of multi-equations,
5. the sum of the lengths of multi-equations, and

6. the number of renaming problems.

Moreover, unification problems that cannot be reduced are in solved form. Clearly, there cannot remain any typing problem since for each construction of the language some rule applies. Similarly, polytypes can always be decomposed. Let us consider a renaming problem $\bar{\alpha} \leftrightarrow \bar{\alpha}'$ for which rule RENAMING-FALSE would not apply. Then variables of $\bar{\alpha}\bar{\alpha}'$ ccould only appear in multi-equations composed of the variables in $\bar{\alpha}\bar{\alpha}'$. Moreover at most one variable of each set $\bar{\alpha}$ and $\bar{\alpha}'$ could appear in each of these multi-equations. Therefore rule RENAMING-TRUE would apply. The remaining rules are standard rules for unification for simple types. ■

Chapter 8

Des classes aux objets par la relation de soustypage

Ce chapitre à été publié dans [111].

Des classes aux objets par la relation de sous-typage

Nous étendons le calcul d'objets primitif d'Abadi et Cardelli avec une opération d'extension sur les objets. Nous enrichissons les types des objets en leur donnant une structure plus précise, flexible et uniforme. Cela permet de typer le sous-typage en largeur et en profondeur simultanément. Les objets peuvent aussi avoir des méthodes virtuelles interdites en lecture et des méthodes co-variantes interdites en écriture. La relation de sous-typage résultante est plus riche et les types des objets peuvent être progressivement affaiblis le long de la relation de sous-typage, passant du niveau des classes au niveau plus traditionnel des objets.

From Classes to Objects via Subtyping

We extend the Abadi-Cardelli calculus of primitive objects with object extension. We enrich object types with a more precise, uniform, and flexible type structure. This enables to type record extension under both width and depth subtyping. Objects may also have extend-only or virtual contra-variant methods and read-only co-variant methods. The resulting subtyping relation is richer, and types of objects can be weaken progressively from a class level to a more traditional object level along the subtype relationship.

8.1 Introduction

Object extension has long been considered unsound when combined with subtyping. The problem may be explained as follows: in an object built with two methods ℓ_1 and ℓ_2 of types τ_1 and τ_2, the method ℓ_1 may require ℓ_2 to be of type τ_2. Forgetting the method ℓ_2 by subtyping would result in the possible redefinition of method ℓ_2 with another, incompatible type τ_3. Then, the invocation of ℓ_1 may fail.

Indeed, the first strongly-typed object-based languages that have been proposed provided either subtyping [1] or object extension [82] to circumvent the problem described above. However, each proposal was missing an important feature supported by the other one.

Both of them were improved later following the same principle: At an earlier stage, object components were assembled in prototypes [81] or classes [2], relying on some extension mechanism to provide inheritance. Objects were formed in a second, atomic step, immediately losing their extension capabilities for ever, to the benefit of subtyping.

In contrast to the previous work, we allow both extension and subtyping at the level of objects, avoiding stratification. Our solution is based on the enrichment of the structure of object types. Thus, our type-system rejects the above counter-example while keeping many other useful programs. In our proposal, an object and its class are unified and can be considered as two different perspectives on the same value: the type of an object is a supertype of the type of its class. Fine grain subtyping allows type information to be lost gradually, both width-wise and depth-wise, slowly fading classes into objects. As is well-known, when more type information is exposed, more operations can be performed (class perspective). On the contrary, hiding a sufficient amount of type information allows for more object interchangeability, but permits fewer operations (object perspective).

We add object extension to the object calculus of Abadi and Cardelli [3]. We adapt their typing rules to our enriched object types. In particular, we force methods to be parametric in self, that is, polymorphic over all possible extensions of the respective object. In this sense, our proposal is not a strict extension of theirs.

In addition to object extension, the enriched type structure has other benefits. We can allow virtual methods in objects (*i.e.* methods that are required by some other method but that have not been defined yet) since we are able to described them in types. Using co-variant subtyping forbids further re-definition of the corresponding method, as in [3]. Since classes are objects, such methods are in fact final methods. Final methods can only be accessed but no more redefined (except, indirectly, by the invocation of a previously defined method).

Virtual methods are useful because they allow objects to be built progressively, component by component, rather than all at once. They also improve security, since they sometime avoid the artificial use of dangerous default methods. While final methods are co-variant, virtual methods, are naturally contra-variant.

Other annotations are also possible. For instance, we are able to tell that a method is *independent*, that is, no method of the object depends on it. Such a method can be hidden, or redefined with a method of any type.

The rest of the paper is organized as follows. In the next section, we describe our solution informally. The following section is dedicated to the formal presentation. In section 8.4, we show some properties of the type system, in particular the type soundness property. Section 8.5 illustrates the gain in security and flexibility of our proposal by running a few examples. To a large extend, these examples can be understood intuitively and may also be read simultaneously with or immediately after the informal presentation. In section 8.6 we discuss possible extensions and variations of our

proposal, as well as further meta-theoretical developments. A brief comparison with other works is done in section 8.7 before concluding.

8.2 Informal presentation

Technically, our first goal is to provide method extension, while preserving some form of subtyping. The counter-example given above does not imply that both method extension and width subtyping are in contradiction. It only shows that combining two existing typing rules would allow to write unsafe programs. Thus, if ever possible, a type system with both method extension and subtyping should clearly impose restrictions when combining them. Our solution is to enrich types so that subtyping becomes traceable, and so that extension can be limited to those fields whose exact type is known.

We first recall record types with symmetric type information. Using a similar structure for object types, some safe uses of subtyping and object extension can be typed, while the counter-example given in the introduction is rejected.

Record types

Record values are partial functions with finite domains that map labels to values. Traditionally, the types of records are also partial functions with finite domains that map labels to types. They are represented as records of types, that is, $\{\ell_i : \tau_i{}^{i\in I}\}$. This type says that fields ℓ_i's are defined with values of type τ_i's. However, it does not imply anything about other fields.

Another richer, more symmetric structure has also been used for record types, originally to allow type inference for records in ML [104, 106]. There, record types are treated as total functions mapping labels to field types, with the restriction that all but a finite number of labels have isomorphic images (*i.e.* are equal modulo renaming). Thus, record types can still be represented finitely by listing all significant labels with their corresponding field types and then adding an extra field-type acting as a template for all other labels.

In their simplest form, field types are either $\mathtt{P}\ \tau$ (read *present* with type τ) or $\mathtt{A}$ (read *absent*). For instance, a record with two fields ℓ_1 of type τ_1 and ℓ_2 of type τ_2 is given type $\langle \ell_1\!:\mathtt{P}\ \tau_1\ ;\ \ell_2\!:\mathtt{P}\ \tau_2\ ;\ \mathtt{A}\rangle$. It could also, equivalently, be given type $\langle \ell_1\!:\mathtt{P}\ \tau_1\ ;\ \ell_2\!:\mathtt{P}\ \tau_2\ ;\ \ell\!:\mathtt{A}\ ;\ \mathtt{A}\rangle$ where ℓ is distinct from ℓ_1 and ℓ_2.

In the absence of subtyping, standard types for records $\{\ell_i : \tau_i{}^{i\in I}\}$ can indeed be seen as a special case of record types, where field variables are disallowed; their standard subtyping relation then corresponds to the one generated by the axiom $\mathtt{P}\ \tau <: \mathtt{A}$ (and obvious structural rules). The type $\{\ell_1 : \tau_1; ..\ell_n : \tau_n\}$ becomes an abbreviation for $\langle \ell_1\!:\mathtt{P}\ \tau_1\ ;\ ..\ell_n\!:\mathtt{P}\ \tau_n\ ;\ \mathtt{A}\rangle$. However, record types are much more flexible. For instance, they inherently and symmetrically express negative information. Before we added subtyping, a field ℓ of type $\mathtt{A}$ was known to be absent in the corresponding record. This is quite different from the absence of information about field ℓ. Such precise information is sometimes essential; a well-known example is record concatenation [50]. Instead of breaking the symmetry with the subtyping axiom $\mathtt{P}\ \tau <: \mathtt{A}$, we might have introduced a new field $\mathtt{U}$ (read *unknown*), with two axioms $\mathtt{P}\ \tau <: \mathtt{U}$ and $\mathtt{A} <: \mathtt{U}$. This would preserve the property that a field of type $\mathtt{A}$ is known to be absent, still allowing present and absent field to be interchanged but at their common supertype $\mathtt{U}$.

Field variables and row variables also increase the expressiveness of record types. However, for simplicity, we do not take this direction here. Below, we use meta-variables for rows. This is just a notational convenience. It does not add any power.

Object types

In their simplest form, objects are just records, thus object types mimic record types. We write object types with $[\rho]$ instead of $\langle\rho\rangle$ to avoid confusion. An object with type $[\ell_1\!:\!\mathtt{P}\ \tau_1\ ;\ \ell_2\!:\!\mathtt{P}\ \tau_2\ ;\ \mathtt{A}]$ possesses two methods ℓ_1 and ℓ_2 of respective types τ_1 and τ_2. Intuitively, an object $[\ell_1 = a_i\ ^{i\in I}]$ can be given type $[\ell_i\!:\!\mathtt{P}\ \tau_i\ ^{i\in I}\ ;\ \mathtt{A}]$ provided methods a_i's have type τ_i's.

However, objects soon differ from records by their ability to send messages to themselves, or to return themselves in response to a method call. More generally, objects are of the form $[\ell_i = \varsigma(x_i)a_i]$. Here, x_i is a variable that is bound to the object itself when the method ℓ_i is invoked. Consistently, the expression a_i must be typed in a context where x_i is assumed of the so-called "mytype", represented by some type variable χ equal to the object type τ. The following typing rule is a variant of the one used in [3].

$$\frac{\tau \equiv \zeta(\chi)[\ell_i\!:\!\mathtt{P}\ \tau_i\ ^{i\in I}\ ;\ \mathtt{A}] \qquad A, \chi = \tau, x_i : \chi \vdash a_i : \tau_i}{A \vdash \zeta(\chi,\tau)[\ell_i = \varsigma(x_i)a_i\ ^{i\in I}] : \tau}$$

(The type annotation (χ,τ) in the object expression binds the name of mytype locally and specifies the type of the object.)

An extendible object v may also be used to build a new object v' with more methods than v and thus of a different type, say τ'. The type τ' of self in v' is different from the type τ of self in v. In order to remain well-typed in v', the methods of v, should have been typed in a context where the type of self could have been τ' as well as τ. This applies to any possible extension v' of v. In other words, methods of an object of type τ should be parametric in all possible types of all possible successive extensions of an object of type τ. This condition can actually be expressed with subtyping by $\chi <: \#\,\tau$, where $\#\,\tau$ is called the *extension type* of τ (also called the internal type of the object). That is, the least upper bound of all exact[1] types of complete extensions (extensions in which no virtual method remains) of objects of external type τ.

A field of type $\mathtt{A}$ can be overridden with methods of arbitrary types. Thus, the best type for that field in the self parameter is $\mathtt{U}$, *i.e.* we choose $\#\mathtt{A}$ to be $\mathtt{U}$. Symmetrically, we choose $\#(\mathtt{P}\ \tau)$ to be $\mathtt{U}$. This makes methods of type $\mathtt{P}\ \tau$ internally unaccessible. Fields of type $\mathtt{P}\ \tau$ are known to be present externally, but are not assumed to be so internally. Thus, fields of type $\mathtt{P}\ \tau$ can be overridden with methods of arbitrary types, such as fields of type $\mathtt{A}$. To recover the ability to send messages to self, we introduce a new type field $\mathtt{R}\ \tau$ (read *required* of type τ). A field of type $\mathtt{R}\ \tau$ is defined with a method of type τ, and is required to remain of at least type τ, internally. Such a field can only be overridden with a method of type τ. Therefore, self can also view it as a field that is, and will remain, of type τ. In math, $\#\mathtt{R}\ \tau$ is $\mathtt{R}\ \tau$. A field of type $\mathtt{P}\ \tau$, can safely be considered as a field of type $\mathtt{R}\ \tau$. Thus, we assume $\mathtt{P}\ \tau <: \mathtt{R}\ \tau$. We also assume $\mathtt{R}\ \tau$ to be a subtype of $\mathtt{U}$. As an example, $\#\,\zeta(\chi)[\ell_1\!:\!\mathtt{R}\ \tau_1\ ;\ \ell_2\!:\!\mathtt{P}\ \tau_2\ ;\ \ell_3\!:\!\mathtt{U}\ ;\ \mathtt{A}]$ is $\zeta(\chi)[\ell_1\!:\!\mathtt{R}\ \tau_1\ ;\ \ell_2\!:\!\mathtt{U}\ ;\ \ell_3\!:\!\mathtt{U}\ ;\ \mathtt{U}]$, or shortly $\zeta(\chi)[\ell_1\!:\!\mathtt{R}\ \tau\ ;\ \mathtt{U}]$.

The extension of a field with a method of type τ requires that field to be either of type $\mathtt{A}$ or $\mathtt{R}\ \tau$ in the original record (the field may also be of type $\mathtt{P}\ \tau$, which is a subtype of $\mathtt{R}\ \tau$.) It is possible to factor the two cases by introducing a new field type $\mathtt{M}\ \tau$ (read *maybe* of type τ), and the axioms $\mathtt{R}\ \tau <: \mathtt{M}\ \tau$, $\mathtt{A} <: \mathtt{M}\ \tau$, and $\mathtt{M}\ \tau <: \mathtt{U}$. Intuitively, $\mathtt{M}\ \tau$ is the union type $\mathtt{R}\ \tau \cup \mathtt{A}$. This allows, in a first step, to ignore the presence of a method while retaining its type, and, in a second step, to forget the type itself. The type of object extension becomes more uniform. Roughly, if the original

[1] The exact type of an object is the type with which the methods can initially be typed. The external type of an object may be a supertype of the exact type.

object has type $[\ell_1{:}\,\mathtt{M}\ \tau_1\ ;\ \tau_2]$ and the new method ℓ_1 has type τ_1 then the resulting object has type $[\ell_1{:}\,\mathtt{R}\ \tau_1\ ;\ \tau_2]$.

A field of type $\mathtt{M}\ \tau$ may later be defined or redefined with some method of type τ, becoming of type $\mathtt{R}\ \tau$, which is a subtype of $\mathtt{M}\ \tau$. It may also be left unchanged and thus remain of type $\mathtt{M}\ \tau$. Thus, a field of type $\mathtt{M}\ \tau$ will always remain of a subtype of $\mathtt{M}\ \tau$. That is, $\#(\mathtt{M}\ \tau)$ is $\mathtt{M}\ \tau$.

Deep subtyping

Subtyping rules described so far allow for width subtyping but not for depth subtyping, since all constructors have been left invariant. The only constructor that could be made covariant without breaking type-soundness is $\mathtt{P}$. Making $\mathtt{R}$ co-variant would be unsafe. However, we can safely introduce a new field type $\mathtt{R}^+\ \tau$ to tell that a method is defined and required to be of a subtype of τ, provided that a field of type $\mathtt{R}^+\ \tau$ is never overridden. On the other hand, a method ℓ_1 can safely be invoked on any object of type $[\ell_1{:}\,\mathtt{R}^+\ \tau_1\ ;\ \mathtt{U}]$, which returns an expression of type τ_1. Of course, we also add $\mathtt{R}\ \tau <: \mathtt{R}^+\ \tau$ to just forget the fact that we are revealing the exact type information.

Symmetrically, a field ℓ of type $\mathtt{M}\ \tau$ cannot be accessed, but it can be redefined with a method of a subtype of τ. Still, it would be unsound to make $\mathtt{M}\ \tau$ contra-variant. By contradiction, consider an object p of type $\zeta(\chi)[\ell{:}\,\mathtt{R}\ \tau\ ;\ \ell'{:}\,\mathtt{P}\ \chi\ ;\ \mathtt{A}]$ where calling method ℓ' overrides ℓ in self with a new method of type τ. By subtyping p_0 could be given type $\zeta(\chi)[\ell{:}\,\mathtt{M}\ \tau_0\ ;\ \ell'{:}\,\mathtt{P}\ \chi\ ;\ \mathtt{A}]$ where τ_0 is a subtype of τ. Then let p_2 of type $\zeta(\chi)[\ell{:}\,\mathtt{M}\ \tau_0\ ;\ \ell'{:}\,\mathtt{P}\ \chi\ ;\ \ell''{:}\,\mathtt{P}\ \mathtt{unit}\ ;\ \mathtt{A}]$ be the extension of p_1 with a new method ℓ'' that requires ℓ of type τ_0. Calling method ℓ' of p_2 restore field ℓ of p_2 to some method of type τ and returns an object p_3. However, calling method ℓ'' of p_3 expects a method ℓ of type τ_0 but finds one of type τ.

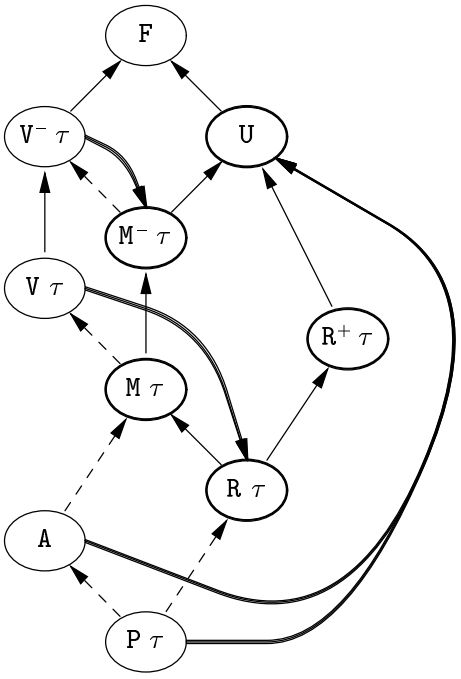
We can still introduce a contra-variant symbol $\mathtt{M}^-$ with the axiom $\mathtt{M}\ \tau <: \mathtt{M}^-\ \tau$. Then, a method $\mathtt{M}^-\ \tau$ can be redefined, but the method in the resulting object remains of type $\mathtt{M}^-\ \tau$ and is thus unaccessible. This is still useful in situations where contra-variance is mandatory or to enforce protection against accidental access (see sections 8.5.6, 8.5.2 and [3].)

Virtual methods

A method ℓ is *virtual* with type τ (which we write $\mathtt{V}\ \tau$) if other methods have assumed ℓ to be of type $\mathtt{R}\ \tau$, while the method itself might not have been defined yet. When an object has a virtual method, no other method of that object can be invoked. Thus, $\mathtt{V}\ \tau$ should not be a subtype of $\mathtt{U}$. A method of type $\mathtt{V}\ \tau$ can be extended as a method of type $\mathtt{R}\ \tau$. Virtual methods may also be contra-variant. We use another symbol $\mathtt{V}^-\ \tau$ to indicate that deep subtyping has been used. A contra-variant virtual method can be extended, but it must remain contra-variant after its extension, *i.e.* of type $\mathtt{M}^-\ \tau$, and thus inaccessible. This may be surprising at first. The intuition if that $\#(\mathtt{V}^-\ \tau)$ should be $\mathtt{R}^-\ \tau$. However, a method of field-type $\mathtt{R}^-\ \tau$ would be inaccessible, since its best type is unknown. Thus $\mathtt{R}^-\ \tau$ has been identified with $\mathtt{M}^-\ \tau$.

For convenience, we also introduce a new constant $\mathtt{F}$ that is a top type for fields. That is, we assume $\mathtt{V}^-\ \tau <: \mathtt{F}$ and $\mathtt{U} <: \mathtt{F}$ (all other relations hold by transitivity).

The final structure of field types and subtyping axioms are summarized in figure 8.1. Thick arrows represent the function #. Thick nodes are used instead of reflexive thick arrows, that is, thick nodes are left invariant by #. Thin arrows represent subtyping. We added a redundant but useful distinction between continuous and dashed thin arrows. They are respectively covariant and contra-variant by type-extension: when a continuous arrow connects τ_1 and τ_2, then $\#\,\tau_1$ is also a subtype of $\#\,\tau_2$; the inverse applies to dashed arrows.



Static		Dynamic		Allowed	
φ	$\#\varphi$	Pre	Type	$\_.\ell$	$\Leftarrow$
P τ	U	$\surd$	$<:\tau$	τ	$\forall$
A	U		$<:\tau$	$\bot$	$\forall$
R τ	φ	$\surd$	$<:\tau$	τ	τ
R$^+$ τ	φ	$\surd$	$<:\tau$	τ	$\bot$
M τ	φ		$<:\tau$	$\bot$	τ
M$^-$ τ	φ		$\forall$	$\bot$	τ
U	φ		$\forall$	$\bot$	$\bot$
V τ	R τ		$<:\tau$	$\bot$	τ
V$^-$ τ	M$^-$ τ		$\forall$	$\bot$	τ

Figure 8.1: Structure of field types

Although it is easy to give intuitions for parts of the hierarchy taken alone (variances, virtual methods, idempotent field-types), we are not able to propose a good intuition for the whole hierarchy. The different components are modular technically, but their intuitive, thus approximative descriptions, cannot be composed here. We think that the field-type hierarchy should be understood locally, and then considered as such.

The table on the right is a summary of field types and their properties. The entry φ in the first column indicates the static external type. The second column $\#\varphi$ is its extension type, *i.e.* the static internal type. The two following columns tell whether the field is guaranteed to be present ($\surd$ sign) and its type if present. The reason for having $<:\tau$ instead of τ is the covariance of P. The symbol $\forall$ means any possible type. The last two columns describe access and overriding capabilities ($\bot$ means disallowed).

8.3 Formal developments

8.3.1 Types

We assume given a denumerable collection of type variables, written α, β, or χ. Type expressions, written with letter τ, are type variables, object types, or the top type T. An object type $\zeta(\chi)[\ell_i{:}\,\varphi_i\,^{i\in I}\,;\varphi]$ is composed of a finite sequence of fields $\ell_i : \varphi_i$, without repetition, and a template φ for fields that are not explicitly mentioned. Variable χ is bound in the object type, and should only appear positively in φ_i's as in φ.

$$\tau ::= \alpha \mid \zeta(\alpha)[\ell_i{:}\,\varphi_i\,^{i\in I}\,;\varphi] \mid \mathtt{T}$$
$$\varphi ::= \mathtt{A} \mid \mathtt{P}\,\tau \mid \mathtt{R}\,\tau \mid \mathtt{M}\,\tau \mid \mathtt{V}\,\tau \mid \mathtt{R}^+\tau \mid \mathtt{M}^-\tau \mid \mathtt{V}^-\tau \mid \mathtt{U} \mid \mathtt{F}$$

The variance of an occurrence is defined in the usual way: it is the parity of the number of times a variable crosses a contra-variant position (*i.e.*, the number of symbols $\mathtt{V}^-$ or $\mathtt{M}^-$) on that path from the root to that occurrence. The set of free variables of τ is $fv(\tau)$. We write $fv^-(\tau)$ the subset of those variables that occurs negatively at least once.

Object types are considered equal modulo reordering of fields. They are also equal modulo expansion, that is, by extracting a field from the template:

$$\zeta(\chi)[\ell_i\colon\varphi_i\,^{i\in I}\,;\,\varphi] = \zeta(\chi)[\ell_i\colon\varphi_i\,^{i\in I}\,;\,\ell\colon\varphi\,;\,\varphi] \qquad \ell \neq \ell_i, \forall i \in I$$

Rules for the formation of types will be defined jointly with subtyping rules in figure 8.2 and are described below.

Notation For convenience and brevity of notation, we use meta-variables ρ for rows of fields, that is, syntactic expressions of the form $(\ell_i\colon\varphi_i\,^{i\in I}\,;\,\varphi_0)$, where φ_i's and I are left implicit. We write $\rho(\ell)$ the value of ρ in ℓ, that is, φ_i if ℓ is one of the ℓ_i's, or φ_0 otherwise. We write $\rho \setminus \ell$ for $(\ell_i\colon\varphi_i\,^{i\in I,\ell_i\neq\ell}\,;\,\varphi_0)$. and $\ell : \varphi; \rho$ for $(\ell\colon\varphi\,;\,\ell_i\colon\varphi_i\,^{i\in I,\ell_i\neq\ell}\,;\,\varphi_0)$. If $\mathcal{R}$ is a relation, we write $\rho\,\mathcal{R}\,\rho'$ for $\forall\ell, \rho(\ell)\,\mathcal{R}\,\rho'(\ell)$.

This is just a meta-notation that is not part of the language of types. It can always be expanded unambiguously into the more explicit notation $(\ell_i\colon\varphi_i\,^{i\in I}\,;\,\varphi)$.

8.3.2 Type extension

We define the *extension* of field type φ, written $\#\,\varphi$ by the two first columns of the table 8.1. Type extension is lifted to object types homomorphically, *i.e.*, $\#\,\zeta(\chi)[\rho]$ is $\zeta(\chi)[\#\,\rho]$. The extension is not defined for type variables, nor for $\mathtt{F}$. Note that the extension is idempotent, that is $\#(\#\,\tau)$ is always equal to $\#\,\tau$.

Well-formation of environments

$$\frac{}{\emptyset \vdash \diamond}\ (\textsc{Env}\ \emptyset) \qquad \frac{E \vdash \tau <: \mathtt{T} \qquad x \notin dom(E)}{E, x : \tau \vdash \diamond}\ (\textsc{Env}\ x) \qquad \frac{E \vdash \tau <: \mathtt{T} \qquad \alpha \notin dom(E)}{E, \alpha <: \tau \vdash \diamond}\ (\textsc{Env}\ \alpha)$$

General subtyping

$$\frac{E, \alpha <: \tau, E' \vdash \diamond}{E, \alpha <: \tau, E' \vdash \alpha <: \tau}\ (\textsc{Sub Var}) \qquad \frac{E \vdash \varphi <: \mathtt{F}}{E \vdash \varphi <: \varphi}\ (\textsc{Sub Ref F}) \qquad \frac{E \vdash \tau <: \mathtt{T}}{E \vdash \tau <: \tau}\ (\textsc{Sub Ref T}) \qquad \frac{E \vdash \tau_1 <: \tau_2 \qquad E \vdash \tau_2 <: \tau_3}{E \vdash \tau_1 <: \tau_3}\ (\textsc{Sub Trans T})$$

$$\frac{E \vdash \varphi_1 <: \varphi_2 \qquad E \vdash \varphi_2 <: \varphi_3}{E \vdash \varphi_1 <: \varphi_3}\ (\textsc{Sub Trans F})$$

Field subtyping (assuming $E \vdash \tau <: \mathtt{T}$)

$$E \vdash \mathtt{P}\ \tau <: \mathtt{A}\ (\textsc{Sub PA}) \qquad E \vdash \mathtt{P}\ \tau <: \mathtt{R}\ \tau\ (\textsc{Sub PR}) \qquad E \vdash \mathtt{A} <: \mathtt{M}\ \tau\ (\textsc{Sub AM}) \qquad E \vdash \mathtt{U} <: \mathtt{F}\ (\textsc{Sub UF}) \qquad E \vdash \mathtt{R}\ \tau <: \mathtt{R}^+\ \tau\ (\textsc{Sub RR}^+) \qquad E \vdash \mathtt{R}\ \tau <: \mathtt{M}\ \tau\ (\textsc{Sub RM})$$

$$E \vdash \mathtt{R}^+\ \tau <: \mathtt{U}\ (\textsc{Sub R}^+\mathtt{U}) \qquad E \vdash \mathtt{M}\ \tau <: \mathtt{V}\ \tau\ (\textsc{Sub MV}) \qquad E \vdash \mathtt{M}\ \tau <: \mathtt{M}^-\ \tau\ (\textsc{Sub MM}^-) \qquad E \vdash \mathtt{M}\ \tau <: \mathtt{U}\ (\textsc{Sub M}^-\mathtt{U}) \qquad E \vdash \mathtt{V}\ \tau <: \mathtt{V}^-\ \tau\ (\textsc{Sub VV}^-)$$

$$E \vdash \mathtt{V}^-\ \tau <: \mathtt{F}\ (\textsc{Sub V}^-\mathtt{F})$$

$$\frac{E \vdash \tau <: \tau'}{E \vdash \mathtt{P}\ \tau <: \mathtt{P}\ \tau'}\ (\textsc{Sub PP}) \qquad \frac{E \vdash \tau <: \tau'}{E \vdash \mathtt{R}^+\ \tau <: \mathtt{R}^+\ \tau'}\ (\textsc{Sub R}^+\mathtt{R}^+) \qquad \frac{E \vdash \tau <: \tau'}{E \vdash \mathtt{M}^-\ \tau' <: \mathtt{M}^-\ \tau}\ (\textsc{Sub M}^-\mathtt{M}^-) \qquad \frac{E \vdash \tau <: \tau'}{E \vdash \mathtt{V}^-\ \tau' <: \mathtt{V}^-\ \tau}\ (\textsc{Sub V}^-\mathtt{V}^-)$$

Object subtyping

$$\frac{E \vdash \diamond}{E \vdash \mathtt{T} <: \mathtt{T}}\ (\textsc{Sub TT}) \qquad \frac{E, \chi <: \mathtt{T} \vdash \rho <: \mathtt{F} \qquad \chi \notin fv^-(\rho)}{E \vdash \zeta(\chi)[\rho] <: \mathtt{T}}\ (\textsc{Sub Obj OK})$$

$$\frac{E \vdash \tau <: \mathtt{T} \qquad E \vdash \tau' <: \mathtt{T} \qquad E, \chi <: \mathtt{T} \vdash \rho <: \rho'}{E \vdash \tau <: \tau'}\ (\textsc{Sub Obj Invariant})\ (\tau \equiv \zeta(\chi)[\rho], \tau' \equiv \zeta(\chi)[\rho'])$$

Figure 8.2: Types and Subtypes

8.3.3 Expressions

Expressions are variables, objects, method invocation, and method overriding.

$$a ::= x \mid \zeta(\chi, \tau)[\ell_i : \varsigma(x_i)a_i] \mid a.\ell \mid a.\ell \Leftarrow \zeta(\chi, \tau)\varsigma(x)a$$

The expression $a.\ell \Leftarrow \zeta(\chi, \tau)\varsigma(x)a_\ell$ is the extension of a on field ℓ with a method $\varsigma(x)a_\ell$. The expression (χ, τ) binds χ to the type of self in a_ℓ and indicates that the resulting type of the extension should be τ. This information is important so that types do not have to be inferred but

only checked. Field update is just a special case of object extension. This is more general, since the selection between update and extension is resolved dynamically.

8.3.4 Well formation of types and subtyping

Typing environments are sequences of bindings written with letter E. There are free kinds of judgments (the second and third ones are similar):

$E ::= \emptyset \mid \alpha <: \tau \mid x : \tau$	Typing environments
$E \vdash \diamond$	Environment E is well-formed
$E \vdash \tau <: \tau'$	Regular type τ is a subtype of τ' in E
$E \vdash \varphi <: \varphi'$	Field type φ is a subtype of φ' in E
$E \vdash a : \tau$	Expression a has type τ in E

The subtyping judgment $E \vdash \tau <: \mathtt{T}$ is used to mean that τ is a well-formed regular type in E, while $E \vdash \varphi <: \mathtt{F}$ means that φ is a well-formed field-type in E. Thus, `T` and `F` also play a role of kinds. For sake of simplicity, we do not allow field variables $\alpha <: \mathtt{F}$ in environments. We have used different meta-variables τ and φ for regular types and field-types for sake of readability, although this is redundant with the constraint enforced by the well-formation rules. The formation of environments is recursively defined with rules for the formation of types and subtyping rules given in figure 8.2.

The subtyping rules are quite standard. Most of the rules are dedicated to field subtyping; they formally described the relation that was drawn in figure 8.1. A few facts are worth noticing. First we cannot derive $E \vdash F <: F$. Thus `F` is only used in $E \vdash \varphi <: \mathtt{F}$ to tell that φ is a well-formed field type. It prevents using `F` in object types. The typing rule Sub `PA` is also worth consideration. By transitivity with other rules, it allows `P` τ to be a subtype of `M` τ', even if types τ and τ' are incompatible. However, it remains true, and this is essential, that `P` τ is a subtype of `R` τ' if and only if τ is a subtype of τ'.

The rule Sub Obj Invariant describes subtyping for object types. As explained above, row variables are just a meta-notation; thus, the judgment $E \vdash \rho <: \rho'$ is just a short hand for $E \vdash \rho(\ell) <: \rho'(\ell)$ for any label ℓ, which only involves a finite number of them. This rule is restrictive and prevents (positive) occurrences of self to be replaced by $\# \tau$ where τ is the current type of the object. In particular, object types cannot be unfolded (see section 8.6.2).

8.3.5 Typing rules

Typing rules are given in figure 8.3. The rules for subsumption, variables, and method invocation are quite standard.

Rule Expr Object has been discussed earlier. The last premise says that the fields ℓ_i may actually be super-types of `P` τ_i in ρ and other fields may also be super types of `A`. One cannot simply require that ρ be $(\ell_i : \mathtt{P}\ \tau_i\ ^{i \in I}; \mathtt{A})$ and later use subsumption, since the assumption made on the type of x_i while typing a_i could then be too weak.

Rule Expr Update is similar to the overriding rule in [3]. This rule is important since it permits both internal and external updates: the result type of the object is exactly the same as the one before the update.

On the contrary, rule Expr Extend is intended to add new methods that were not necessarily defined before, and thus change the type of the object. There are three different sub-cases in rule Expr Extend; the one that applies is uniquely determined by the given type τ. Then the type of field ℓ in the argument is deduced from the small table.

$$\text{(Expr Subsumption)}\quad \frac{E \vdash a : \tau \qquad E \vdash \tau <: \tau'}{E \vdash a : \tau'} \qquad \text{(Expr Var)}\quad \frac{E, x : \tau, E' \vdash \diamond}{E, x : \tau, E' \vdash x : \tau}$$

$$\text{(Expr Object)}\ (\tau \equiv \zeta(\chi)[\rho])\quad \frac{E, \chi <: \#\,\tau, x_i : \chi \vdash a_i : \tau_i, i \in I \qquad E, \chi <: \#\,\tau \vdash (\mathtt{P}\ \tau_i{}^{i\in I}; \mathtt{A}) <: \rho}{E \vdash \zeta(\chi, \tau)[\ell_i = \varsigma(x_i)a_i{}^{i\in I}] : \tau}$$

$$\text{(Expr Select)}\quad \frac{E \vdash a : \tau \qquad E \vdash \tau <: \zeta(\chi)[\ell{:}\,\mathtt{R}^+\ \tau_\ell\ ;\ \mathtt{U}]}{E \vdash a.\ell : \tau_\ell\{\tau/\chi\}}$$

$$\text{(Expr Update)}\quad \frac{E \vdash a : \tau \qquad E \vdash \tau <: \zeta(\chi)[\ell{:}\,\mathtt{R}\ \tau_\ell\ ;\ \rho_0] \qquad E, \chi <: \#\,\tau, x : \chi \vdash a_\ell : \tau_\ell}{E \vdash a.\ell \Leftarrow \zeta(\chi, \tau)\varsigma(x)a_\ell : \tau}$$

$$\text{(Expr Extend)}\ (\tau \equiv \zeta(\chi)[\rho])\quad \frac{(\varphi_0, \rho(\ell)) \in \{(\mathtt{A}, \mathtt{P}\ \tau_\ell), (\mathtt{V}\ \tau_\ell, \mathtt{R}\ \tau_\ell), (\mathtt{V}^-\ \tau_\ell, \mathtt{M}^-\ \tau_\ell)\} \qquad E \vdash a : \zeta(\chi)[\ell{:}\,\varphi_0\ ;\ \rho \setminus \ell] \qquad E, \chi <: \#\,\tau, x : \chi \vdash a_\ell : \tau_\ell}{E \vdash a.\ell \Leftarrow \zeta(\chi, \tau)\varsigma(x)a_\ell : \tau}$$

Figure 8.3: Typing rules

Rules Expr Extend and Expr Update both apply only when τ is of the form $\zeta(\chi)[\ell{:}\,\mathtt{P}\ \tau_\ell\ ;\ \rho]$ or $\zeta(\chi)[\ell{:}\,\mathtt{R}\ \tau_\ell\ ;\ \rho]$. Then, the requirements on the type of a are the same (letting the premise of Sub Extend be preceded by a subsumption rule). Thus, different derivations lead to the same judgment. It would also be possible to syntactically distinguish between object extension and method update, as well as to separate the extension between three different primitive corresponding to each of the three typing cases.

8.3.6 Operational semantics

We give a reduction semantics for a call-by-value strategy. Values are reduced to objects. A leftmost outermost evaluation strategy is enforced by the evaluation contexts C.

$$v ::= \zeta(\chi, \tau)[\ell_i = \varsigma(x_i)a_i{}^{i\in I}] \qquad\qquad C ::= \{\} \mid C.\ell \mid C.\ell \Leftarrow \zeta(\tau, \chi)\varsigma(x)a$$

The reduction rules are given in figure 8.4. Since programs are explicitly typed, the reduction must also manipulate types in order to maintain programs both well-formed and well-typed, even though it is not type-driven. In fact, the reduction uses an auxiliary binary operation on types $\varphi \Leftarrow \varphi'$, to recompute the witness type of object values during object extension. It is defined in figure 8.5. The partial $\varphi \Leftarrow \varphi'$ is extended to object types homomorphically, *i.e.*, $\zeta(\chi)[\rho] \Leftarrow \zeta(\chi)[\rho']$ is $\zeta(\chi)[\rho \Leftarrow \rho']$. Type extension is defined so as it validates lemma 40. When there is some flexibility, we sought for more uniformnity. Type extension is undefined when the cell is left empty in the figure. Those are cases that will never meet the hypotheses of lemma 40.

Let ℓ and ${\ell_i}^{i\in I}$ be distinct labels, j in I,
and v be of the form $\zeta(\chi,\tau)[\ell_i = \varsigma(x_i)a_i{}^{i\in I}]$.

$v.\ell_j \longrightarrow a_j\{\tau/\chi\}\{v/x\}$ (SELECT)

$v.\ell_j \Leftarrow \zeta(\chi,\tau')\varsigma(x)a \longrightarrow \zeta(\chi,\tau \Leftarrow \tau')[\ell_i = \varsigma(x_i)a_i{}^{i\in I-j}, \ell_j = \varsigma(x)a]$ (UPDATE)

$v.\ell \Leftarrow \zeta(\chi,\tau')\varsigma(x_0)a_0 \longrightarrow \zeta(\chi,\tau \Leftarrow \tau')[\ell_i = \varsigma(x_i)a_i{}^{i\in I}, \ell = \varsigma(x)a]$ (EXTEND)

if $a_1 \longrightarrow a_2$ then $C\{a_1\} \longrightarrow C\{a_2\}$ (CONTEXT)

Figure 8.4: Reduction rules

$\Leftarrow$		φ'							$\#\,\varphi$
		$\mathtt{P}\,\tau',\ \mathtt{A}$	$\mathtt{U}$	$\mathtt{R}\,\tau',\ \mathtt{R}^+\,\tau'$	$\mathtt{M}\,\tau'$	$\mathtt{M}^-\,\tau'$	$\mathtt{V}\,\tau'$	$\mathtt{V}^-\,\tau'$	
φ	$\mathtt{P}\,\tau,\ \mathtt{A}$	φ'							$\mathtt{U}$
	$\mathtt{R}\,\tau,\ \mathtt{R}^+\,\tau,\ \mathtt{U},\ \mathtt{M}^-\,\tau$	.	φ						φ
	$\mathtt{M}\,\tau$	.	.	$\mathtt{R}\,\tau$	φ		$\mathtt{V}\,\tau$	φ	φ
	$\mathtt{V}\,\tau$	.	.	$\mathtt{R}\,\tau$			φ		$\mathtt{R}\,\tau$
	$\mathtt{V}^-\,\tau$	.	.	.	.	$\mathtt{M}^-\,\tau$	.	φ	$\mathtt{M}^-\,\tau$
$\#\,\varphi'$		$\mathtt{U}$	φ'				$\mathtt{R}\,\tau'$	$\mathtt{M}^-\,\tau'$	

Figure 8.5: Type reduction $\varphi \Leftarrow \varphi'$

8.4 Soundness of the typing rules

The soundness of the typing rules results from a combination of subject reduction and canonical forms. The proof of subject reduction is standard (see [3] for instance). A few classical lemmas help simplifying the main proof.

Lemma 37 (Bound weakening) *If $E \vdash \tau <: \tau'$ and $E, \alpha <: \tau', E' \vdash \mathcal{J}$, then $E, \alpha <: \tau, E' \vdash \mathcal{J}$.*

Proof: By induction on the size of the proof of the derivation of the second. ■

Lemma 38 (Substitution)

1. *If $E, \alpha <: \tau, E' \vdash \mathcal{J}$ and $E \vdash \tau' <: \tau$, then $E, E'\{\tau'/\alpha\} \vdash \mathcal{J}\{\tau'/\alpha\}$.*
2. *If $E, x : \tau, E' \vdash \mathcal{J}$ and $E \vdash a : \tau$, then $E, E' \vdash \mathcal{J}\{a/x\}$.*

Lemma 39 (Structural subtyping)

1. *If $\tau \equiv \zeta(\chi)[\rho]$ and $E \vdash \tau <: \tau'$, then τ' is either $\mathtt{T}$ or of the form $\zeta(\chi)[\rho']$ and $E, \chi <: \mathtt{T} \vdash \rho <: \rho'$.*
2. *If $E \vdash \varphi <: \mathtt{R}\,\tau_\ell$, then φ is either $\mathtt{R}\,\tau_l$ or $\mathtt{P}\,\tau_0$ where $E \vdash \tau_0 <: \tau_\ell$.*

3. *If* $E \vdash \varphi <: \mathtt{R}^+\ \tau_\ell$, *then* φ *is either* $\mathtt{P}\ \tau_0$, $\mathtt{R}\ \tau_0$, *or* $\mathtt{R}^+\ \tau_0$ *where* $E \vdash \tau_0 <: \varphi_\ell$.

4. *If* $E \vdash \varphi <: \mathtt{P}\ \tau_\ell$, *then* φ *is* $\mathtt{P}\ \tau_0$ *where* $E \vdash \tau_0 <: \tau_\ell$.

Etc.

Proof: By induction on the size of subtyping derivations. Should use the fact that transitivity rules can be pushed to the leaves. ■

The proof of subject reduction also uses an essential lemma that relates computation on types to subtyping. Actually, the proof does not depend on the particular definition of #, but only on the following lemma.

Lemma 40 (Type computation) *Let* τ *and* τ' *be two object types* $\zeta(\chi)[\rho]$ *and* $\zeta(\chi)[\rho']$. *Assume that there exists a row* ρ'' *such that* $E, \chi <: \mathtt{T} \vdash \rho <: \rho''$ *and for each label* ℓ, *the pair* $(\rho''(\ell), \rho'(\ell))$ *is one of the four forms* $(\mathtt{A}, \mathtt{P}\ \tau_\ell)$, $(\mathtt{V}\ \tau_\ell, \mathtt{R}\ \tau_\ell)$, $(\mathtt{V}^-\ \tau_\ell, \mathtt{M}^-\ \tau_\ell)$, *or* (φ, φ). *Let* $\hat\tau$ *be* $\tau \Leftarrow \tau'$ *and* $\hat\rho$ *be* $\rho \Leftarrow \rho'$. *Then,*

$$E \vdash \hat\tau <: \tau' \qquad\qquad E \vdash \#\,\hat\tau <: \#\,\tau \qquad\qquad E \vdash \#\,\hat\tau <: \#\,\tau'$$

Moreover, in the three first cases, if $E, \chi <: \mathtt{T} \vdash \rho(\ell) <: \rho'(\ell)$, *then* $E, \chi <: \mathtt{T} \vdash \rho(\ell) <: \hat\rho(\ell)$; *otherwise* $E, \chi <: \mathtt{T} \vdash \mathtt{P}\ \tau_\ell <: \hat\rho(\ell)$.

The proof can be found in the appendix 8.9.

Lemma 41 (Virtual methods) *If* $E \vdash \tau <: \zeta(\chi)[\mathtt{U}]$, *then* $E \vdash \tau <: \#\,\tau$.

Proof: This is obviously true field by field: the only field that does not satisfy $E \vdash \tau <: \#\,\tau$ are virtual fields, which are excluded if $E \vdash \tau <: \mathtt{U}$. The property easily follows for object types. ■

Theorem 5 (Subject Reduction) *Typings are preserved by reduction. If* $E \vdash a : \tau_a$ *and* $a \longrightarrow a'$ *then* $E \vdash a' : \tau_a$.

The proof can be found in the appendix 8.4.

Theorem 6 (Canonical Forms) *Well-typed expressions that cannot be reduced are values. If* $\emptyset \vdash a : \tau$ *and there exists no* a' *such that* $a \longrightarrow a'$, *then* a *is a value.*

Proof: If a value v has type $\zeta(\chi)[\ell\colon \mathtt{P}\ \tau\ ;\ \mathtt{U}]$, then v must have a field ℓ. The theorem is then a trivial induction on the size of a, assuming that a cannot be reduced. ■

8.5 Examples

For simplicity, we assume that the core calculus has been extended with abstraction and application. This extension could either be primitive or derived from the encoding given in section 8.5.6. For brievity, we write $a.\ell \Leftarrow a'$ instead of $a.\ell \Leftarrow \zeta(\chi, \tau)\varsigma(z)a'$ when a' does not depend on the self parameter z. In practice, other abbreviations could be made, but we avoid them here to reduce confusion.

We consider the simple example of points and colored points. These objects can of course already be written in [3]. The expressiveness of our calculus is not so much its capability to write new forms of complete objects but to provide new means of defining them. This provides more flexibility, increases security in several ways (see parts 8.5.2, 8.5.3, and 8.5.6), and removes the complexity of the encoding of classes into objects.

8.5.1 Objects

A point object p_0 can be defined as follows:

$$\zeta(\chi, \texttt{point})[x = 0\ ;\ mv = \varsigma(z)\lambda y.(z.x \Leftarrow y)\ ;\ print = \varsigma(z)print\_int\ z.x]$$

where `point` is $\zeta(\chi)[x{:}\,\texttt{R int}\ ;\ mv{:}\,\texttt{P int} \to \chi\ ;\ print{:}\,\texttt{P unit}\ ;\ \texttt{A}]$. As in [3], new points can be created using method update as in $p_0.x \Leftarrow \zeta(\chi, \texttt{point})\varsigma(z)1$. Moreover, colored points can be defined inheriting from points:

$$\texttt{cpoint} \triangleq \zeta(\chi)[x{:}\,\texttt{R int}\ ;\ c{:}\,\texttt{R bool}\ ;\ mv{:}\,\texttt{P int} \to \chi\ ;\ print{:}\,\texttt{P unit}\ ;\ \texttt{A}]$$

$$cp \triangleq (p_0.c \Leftarrow \zeta(\chi, \texttt{cpoint})\varsigma(z)\texttt{true}).print \Leftarrow \zeta(\chi, \texttt{cpoint})\varsigma(z)\texttt{if}\ z.c\ \texttt{then}\ print\_int\ z.x$$

When two values of different types have a common super-type τ, they can be interchanged in any context that expects a term of type τ. Here, `cpoint` is not a subtype of `point`, since both types carry too precise type information. However, they admit the common super-type $\zeta(\chi)[x{:}\,\texttt{R int}\ ;\ c{:}\,\texttt{U}\ ;\ mv{:}\,\texttt{P int} \to \chi\ ;\ print{:}\,\texttt{P unit}\ ;\ \texttt{A}]$..

8.5.2 Abstraction via subtyping

Subtyping can also be used to enforce security. For instance, field x may be hidden by weakening its type to `U`. Similarly, method mv may be protected against further redefinition by weakening its type to $\texttt{R}^+\,\tau$. That is, by giving p_0 the type $\zeta(\chi)[mv{:}\,\texttt{R}^+\texttt{int} \to \chi\ ;\ print{:}\,\texttt{R unit}\ ;\ \texttt{U}]$. While method mv can no longer be directly redefined, there is still a possibility for indirect redefinition. For instance, method $print$ could have been written so that it overrides method mv before printing. To ensure that a method can never be redefined, directly or indirectly, it must be given type $\texttt{R}^+\,\tau$ at its creation.

8.5.3 Virtual methods

The creation of new points by updating the field of an already existing point is not quite satisfactory since it requires the use of default methods to represent the undefined state, which are often arbitrary and may be a source of errors. Indeed, a class of points can be seen as a virtual point lacking its field components.

$$\texttt{POINT} \triangleq \zeta(\chi)[x{:}\,\texttt{V int}\ ;\ mv{:}\,\texttt{P int} \to \chi\ ;\ print{:}\,\texttt{P unit}\ ;\ \texttt{A}]$$

$$P \triangleq \zeta(\chi, \texttt{POINT})[mv = \varsigma(z)\lambda y.(z.x \Leftarrow y)\ ;\ print = \varsigma(z)print\_int\ z.x]$$

New points are then created by filling in the missing fields:

$$\texttt{new\_point} \triangleq \lambda y.(P.x \Leftarrow \zeta(\chi, \texttt{point})\varsigma(z)y) \qquad p_1 \triangleq \texttt{new\_point}\ 0$$

8.5.4 Traditional class-based perspective

To keep closer to the traditional approach, we may by default choose to hide both fields corresponding to instance variables and the extendible capabilities of the remaining methods. For instance, treating x as an instance variable, and mv and $print$ as "regular" methods, we choose $\zeta(\chi)[mv{:}\,\texttt{R}^+\texttt{int} \to \chi\ ;\ print{:}\,\texttt{R}^+\texttt{unit}\ ;\ \texttt{U}]$ for `point`. Intuitively, the object-type `point` hides all information that is not necessary to increase security. Conversely, the class-type `POINT` remains as

precise as possible, to keep expressiveness. Indeed, a class of points is still an object. However, as opposed to the previous section, we adopt some uniform, more structured style, treating "real" objects differently from those representing classes.

In colored points, we may choose to leave field c readable and overridable, as if we defined two methods *set_c* and *get_c*.

$$\texttt{cpoint} \stackrel{\triangle}{=} \zeta(\chi)[c{:}\,\texttt{R bool} \,;\, mv{:}\,\texttt{R}^{+}\texttt{int} \to \chi \,;\, print{:}\,\texttt{R}^{+}\texttt{unit} \,;\, \texttt{U}]$$

Single inheritance is obtained by class extension:

$$\texttt{CPOINT} \stackrel{\triangle}{=} \zeta(\chi)[x{:}\,\texttt{V int} \,;\, c{:}\,\texttt{V bool} \,;\, mv{:}\,\texttt{P int} \to \chi \,;\, print{:}\,\texttt{P unit} \,;\, \texttt{A}]$$

$$CP \stackrel{\triangle}{=} (P.print \Leftarrow \zeta(\chi, \texttt{CPOINT})\varsigma(z)\texttt{if } z.c \texttt{ then } print\_int\ z.x)$$

$$\texttt{new\_cpoint} \stackrel{\triangle}{=} \lambda y.\lambda w.(CP.x \Leftarrow \zeta(\chi, \texttt{cpoint})\varsigma(z)y).c \Leftarrow \zeta(\chi, \texttt{cpoint})\varsigma(z)w q_0 \stackrel{\triangle}{=} \texttt{new\_cpoint}\ 0\ \texttt{true}$$

While `CPOINT` is not a subtype of `POINT` at the class level, we recover the usual relationship that `cpoint` is a subtype of `point` at the object level. Moreover, at the object level, types are invariant by #. Thus, we also recover the subtyping relation of [3]. In particular, object types can be unfolded. For example,

$$\texttt{cpoint} <: \zeta(\chi)[c{:}\,\texttt{R bool} \,;\, mv{:}\,\texttt{R}^{+}\texttt{int} \to \texttt{point} \,;\, print{:}\,\texttt{R}^{+}\texttt{unit} \,;\, \texttt{U}]$$

8.5.5 An advanced example

A colorable point p' is a point prepared to be colored without actually being colored. It can be obtained by adding to the point p_0 an extra method `paint` that when called with an argument `y` returns the colored point obtained by adding the color field c with value y and by updating the print method of p_0.

$$p' \stackrel{\triangle}{=} p_0.paint \Leftarrow \varsigma(z, \texttt{point}')\lambda y.\\ \quad ((z.c \Leftarrow y).print \Leftarrow \zeta(\chi, \texttt{cpoint})\varsigma(z)\texttt{if } z.c \texttt{ then } print\_int\ z.x)$$

where $\texttt{point}'$ is

$$\zeta(\chi)[x{:}\,\texttt{R int} \,;\, mv{:}\,\texttt{R}^{+}\texttt{int} \to \chi \,;\, print{:}\,\texttt{R unit} \,;\, paint{:}\,\texttt{P bool} \to \texttt{cpoint} \,;\, c{:}\,\texttt{M bool} \,;\, \texttt{U}]$$

This example may be seen as the installation (method *paint*) of a new behavior (method *print*) that interacts with the existing state x and adds some new state c. The above solution becomes more interesting if each installation involves many methods, and especially if several installation are either different fields of the same objects or the same field of different objects. Then, the installation procedure can be selected dynamically by message invocation instead of manually by applying an external function to the object.

8.5.6 Encoding of the lambda-calculus

This part improves the encoding proposed in [3]. It also illustrates the use of virtual methods and variances. The untyped encoding of the lambda-calculus into objects in [3] is the following[2]:

$$\langle\!\langle x \rangle\!\rangle \triangleq x.\mathtt{arg} \qquad\qquad \langle\!\langle \lambda x.M \rangle\!\rangle \triangleq [\mathtt{arg} = \varsigma(x)x.\mathtt{arg}\ ;\ \mathtt{val} = \varsigma(x).\langle\!\langle M \rangle\!\rangle]$$

$$\langle\!\langle M\ M' \rangle\!\rangle \triangleq (\langle\!\langle M \rangle\!\rangle.\mathtt{arg} \Leftarrow \varsigma(x)\langle\!\langle M' \rangle\!\rangle).\mathtt{val}$$

A function is encoded as an object with a diverging method `arg`. The encoding of an application overrides the method `arg` of the encoding of the function with the encoding of the argument and invokes the method `val` of the resulting object. Programs obtained by the translation of functional programs will never call `val` before loading the argument. However, if the encoding is used as a programming style, the type system will not provide as much safety as a type system with primitive function types would. The method `val` could also be called, accidently, before the field `arg` has been overridden. In general, this will, in turn, call the method `arg` and diverge. The use of default diverging methods is a hack that palliates the absence of virtual methods. It can be assimilated to a "method not understood" type error and one could argue that the encoding of [3] is not strongly typed.

The encoding can be improved using object extension to treat a function $\lambda x.M$ as an object $[\mathtt{val} = \varsigma(x).\langle\!\langle M \rangle\!\rangle]$ with a virtual method `arg` (remember that $x.\mathtt{arg}$ may appear in $\langle\!\langle M \rangle\!\rangle$). The type-system will then prevent the method `val` to be called before the argument has been loaded. More precisely, let us consider the simply typed lambda-calculus:

$$t ::= \alpha \mid t \to t \qquad\qquad M ::= x \mid \lambda x : t.M \mid M\ M$$

Functional types are encoded as follows:

$$\langle\!\langle \alpha \rangle\!\rangle \triangleq \alpha \qquad\qquad \langle\!\langle t \to t' \rangle\!\rangle \triangleq \zeta(\chi)[\mathtt{arg}\colon \mathtt{V}^-\langle\!\langle t \rangle\!\rangle\ ;\ \mathtt{val}\colon \mathtt{R}^+\langle\!\langle t' \rangle\!\rangle\ ;\ \mathtt{U}]$$

This naturally induces a subtyping relation between function types that is contra-variant on the domain and covariant on the co-domain. The typed encoding is given by the following inference rules:

$$\frac{x : t \in A}{A \vdash x : t \Rightarrow x.\mathtt{arg}} \qquad \frac{A, x : t \vdash M : t' \Rightarrow a \qquad x \notin dom\,(A)}{A \vdash \lambda x : t.M : t \to t' \Rightarrow \zeta(\chi, \langle\!\langle t \to t' \rangle\!\rangle)[\mathtt{val} = \varsigma(x).a]}$$

$$\frac{A \vdash M : t' \to t \Rightarrow a \qquad A \vdash M' : t' \Rightarrow a'}{A \vdash M\ M' : t \Rightarrow (a.\mathtt{arg} \Leftarrow \zeta(\chi, \#\langle\!\langle t \to t' \rangle\!\rangle)\varsigma(x)a').\mathtt{val}}$$

It is easy to see that the translation transforms well-typed judgments $\emptyset \vdash M : t$ into well-typed judgments $\emptyset \vdash \langle\!\langle M \rangle\!\rangle : \langle\!\langle t \rangle\!\rangle$.

As in [3], the translation provides a call-by-name operational semantics for the lambda-calculus. The encoding of [3] also provides an equational theory for the object calculus and, thefore, for the lambda calculus, via translation, which we do not.

[2] If both functions and objects co-exist, one should actually mark variables introduced by the encoding of functions so as to leave the other variables unchanged.

8.6 Discussion

8.6.1 Variations

Several variations can be made by consistently modifying field-types, their subtyping relationship, and the typing rule for object extension. The easiest is to drop some subtyping asumption (such as SUB PP, or SUB PA) or drop the field-type $\mathtt{P}\ \tau$ altogether. This weakens the type system (some examples are not typable any longer), but it retains the essential features. More significant simplifications can be made at the price of a higher restriction of expressiveness. For instance, virtual field-types could be removed.

Some extensions or modifications to the type hierarchy are also possible. For instance, one could introduce fields of type $\dagger\mathtt{P}\ \tau$ that do no depend on any other method. These methods would be dual of those of type $\mathtt{P}\ \tau$ on which no other method depend; somehow they would behave as record fields in the sense they could always be called even if the object is virtual. This extends to field-types $\dagger\mathtt{R}\ \tau$ and $\dagger\mathtt{R}^+\ \tau$ similarly.

8.6.2 Better subtyping for object types

The subtyping rule SUB-OBJ-INVARIANT does not allow unfolding of object types. It is thus weaker than the Abadi-Cardelli:

$$\frac{\begin{array}{l}(\text{SUB OBJ DEEP})\ (\tau \equiv \zeta(\chi)[\rho], \tau' \equiv \zeta(\chi)[\rho']) \\ E \vdash \tau <: \mathtt{T} \qquad E \vdash \tau' <: \mathtt{T} \qquad E, \chi <: \tau \vdash \rho <: \rho'\end{array}}{E \vdash \tau <: \tau'}$$

This rule would not be correct, since it would not be transitive. Indeed transitivity would require that $A \vdash \tau <: \tau'$ implies $A \vdash \#\,\tau <: \#\,\tau'$ which is not true.

Just replacing the bound $\mathtt{T}$ of χ in SUB OBJ DEEP by $\#\,\tau$ would actually not behave well with respect to transitivity. In a preliminary version of this work [112], we added another premise to recover transitivity. However, this simultaneously weakens the subtyping relationship, and some useful examples become untypable.

It should be possible to define a subtyping rule that allows unfolding of object types only when there are no more extension capabilities. It seems however, that the subtyping structure of fields should either be simplified (eliminating the arrow from $\mathtt{M}\ \tau$ to $\mathtt{V}\ \tau$) or enriched (e.g. avoid $\mathtt{M}\ \tau <: \mathtt{V}\ \tau$ but only once in a certain definite state).

8.6.3 Extensions

Imperative update is an orthogonal issue to the one studied here, and it could be added without any problem. Object extension should, of course, remain functional.

Equational theory We see no difficulty in adding an equational theory to our calculus, but this remains to be investigated. Treating object extension as a commutative operator would allow to reduce object construction to a sequence of object extensions of the empty object (virtual methods would be crucial here).

Higher-order types As shown above, our objects are sufficiently powerful to represent classes. As opposed to [3], this does not necessitate higher-order polymorphism because methods are already required to be parametric in all possible extensions of self.

The addition of higher-order polymorphism might still be useful, in particular to enable parametric classes. We believe that there is no problem in constraining type abstraction by some supertype bound, written $\alpha <: \tau$ as in $F_{<:}$. However, it would also be useful to introduce #-bounds of the form $\alpha <: \# \tau$. This might require more investigation.

Row variables and binary methods We have used row variables only as a meta-notation for simplifying the presentation. It would be interesting to really allow row variables in types. This would probably augment the expressiveness of the language, since it should provide some form of matching that revealed quite useful, especially for binary methods [18, 16, 113].

Actually, it remains to investigate how the presented calculus could be extended to cope with binary methods. Row variables might not be sufficient to express matching, and some new form of matching might have to be found. It is unclear whether the known solutions [17] could be adapted to our calculus.

In this section, we review extensions and future works.

8.6.4 Imperative calculus

In our proposal objects are functional. This is, of course, the harder case. Adding imperative operations should not be a problem. The overriding primitive may be given an imperative semantics. However, the extension should remain functional, since it changes types.

8.6.5 Equational theory

We see no difficulty in adding an equational theory to the calculus, following [3]. The encoding of the lambda-calculus given in section 8.5.6 should validate the α and β equalities.

The addition of an equational theory would be particularly interesting for our proposal because objects would then become equal to a sequence of extensions, modulo re-ordering, and thus could be removed from the calculus (only the empty object need to be kept.)

8.6.6 Higher-order types, row variables, matching, and binary methods

As shown above, our objects are sufficiently powerful to represent classes. As opposed to [3], this does not necessitate higher-order polymorphism because methods are already required to be parametric in all possible extensions of self.

The addition of higher-order polymorphism might still be useful, in particular to enable parametric classes. We believe that higher-order polymorphism could be easily added. Bounded quantification allows to constrain type variables by some bound τ, written $\alpha <: \tau$ so that they can only be instantiated by subtypes of the τ. It might also be useful to constrain type variables by #-bounds written $\alpha \# <: \tau$ so that they can only be instantiated by types whose type-extension is a subtype of τ. This is certainly a more difficult task.

Rows have been used as a meta-notation, but field and row variables have been carefully avoided for simplicity. Having field and row variables in the language of types would turn the meta-notation into an internal concept. Furthermore, it would increase expressiveness, especially when combined with polymorphism. Again, variable type-extension bounds might be necessary to obtain the whole benefit from field and row variables.

To illustrate some of the difficulties, consider the following example. It would be interesting to give the function `new_point` a polymorphic type; then, it could be reused in the definition of

`new_cpoint`. The function `new_point` could be defined as:

$$\forall\alpha <: \mathtt{T}.\forall\rho <: \mathtt{T} \Rightarrow \mathtt{F}.$$
$$\lambda P : \zeta(\chi)[x{:}\,\mathtt{V}\ \alpha\ ;\ \rho(\chi)].\lambda y : \alpha.(P.x \Leftarrow \zeta(\chi, \zeta(\chi)[x{:}\,\mathtt{P}\ \alpha\ ;\ \rho(\alpha)])\varsigma(z)y)$$

Formally, this requires, however, a lot of machinery: higher-order abstraction, type operators, row variables, and variable type-extension bounds. Moreover, this is one of the simplest example, since here, the new method does not uses self.

In fact, type extension allows to express abstraction over all (types of all) objects that extend an object of the current type; in other words over all (types of) objects of an instance of a subclass of the class the current object. This is clearly related to the notion of matching [18, 16]. While matching with simple object types can advantageously be replaced, or simulated, by row variables as in Objective ML [113], row variables are likely to be insufficient to provide the essence of matching with the present enriched object types.

Matching and row variables may solve binary methods in some simpler object calculus. It is not clear yet whether the use of matching, row variables, higher-order subtyping as in [3], or any of the known solutions [17] could be adapted to our calculus. This is one of the most important investigations to pursue.

8.6.7 Encoding of objects

It remains future work to find a good encoding of our objects into a typed lambda-calculus with records. Our approach and many intuitions were in fact motivated by the self-application interpretation of objects as records of functions [3]. This interpretation can model most operations on object-based approaches, but delegation-based inheritance in an untyped calculus with records. Unfortunately, in any known type system there is always some operation that cannot be typed. Richer object types have made it possible to unify objects with classes. Richer record types are certainly needed to see objects for what they really are—records of functions.

Methods are usually polymorphic in classes but they are specialized at the creation of objects, whether the objects are primitive [3] or encoded [60]. Our methods are parametric in self inside objects. This is a major difference that might help find other encodings that those proposed in [60].

8.7 Comparison with other works

Our proposal is built on the calculus of objects of Abadi and Cardelli [3], which is invoked throughout this paper. Our use of variance annotations is in principle similar to theirs. By attaching variance annotations to field-types rather than to fields themselves, we eliminate some useless types such as $\mathtt{M}^{+}\ \tau$. Indeed, such a field could not be overriden, nor accessed, and thus it could be just given type `U`. (Our use of variances also eliminate the ability to specify the type of a field without specifying its variance, which may cause problem with type inference [88].) An essential imported tool is the structure of record-types of [104], which was originally designed for type inference in ML [106]. The use of a richer structure of record types has previously been proposed for type checking records [49, 50, 27, 25]. To our knowledge, the benefits of symmetric information were first transfered from record types to object types in [109]. There, first-order typing rules for objects with extension and both deep and width subtyping were roughly drafted without any formal treatment.

A similar approach has also been independently proposed by Bono, Liquori and others. Their first related work [11] has later lead to many closely related proposals [13, 12, 9, 75, 74, 76]. Most of these are extensions of the Fisher-Honsel calculus of objects [81]. The differences between their approach and the one of [3] (which is also ours) are not always significant but they make a close comparisson more difficult. Only two of these works [76, 75] are extensions of the Abadi-Cardelli calculus of objects [3] and are thus more connected to our proposal. The first-order version [76], is subsumed by both [109] (which also covers deep subtyping) and [75] (which also addresses self types.) Our proposal extends both [109] and [75].

The most interesting comparison can be made with [75]. The main motivation and the key idea behind both proposals are similar: they integrate object subtyping and object extension, using a richer type structure to preserve type soundness. Saturated *vs* diamond types correspond to our object-types with a field template $\mathtt{U}$ *vs* $\mathtt{A}$, respectively. Our treatment seems more uniform. We only have one kind of object types. We distinguish between the "saturation" and "diamond" properties in fields instead of objects. As a result, we can write an object type that is saturated, except for a few particular fields. Our proposal also includes several additional features: it addresses deep subtyping and virtual methods; it also allows methods to extend self. Moreover, in our proposal, the subtyping relationship is structural for object types. Additionally, subtyping axioms are only given at the level of fields, each one of them treating a different important subtyping capability. As a result, object types have a more regular structure, and can easily be adapted to further extensions. We think this is easier to understand, to modify, and to manipulate.

An alternative to virtual methods has also been studied in [9], using a quite different approach, which consists in annotating each method with the list of all other methods they depend on. Thus, each method has a different view of self. Their approach to incomplete objects is, in principle, more powerful that ours; in particular, they can type programs that even traditional class-based languages would reject. We found their types of objects too detailed, and thus their proposal less practical than ours. (Tracing dependencies is closer to some form of program analysis than to standard type systems.) In fact, we intendedly restricted our type system so that methods have a uniform view of self. In practice, our solution is sufficient to capture common forms of inheritance.

In [81], pre-objects have pro-types and can be turned into objects with obj-types by subtyping. Pro-types and obj-types are similar to our object types $\zeta(\chi)[\ell_i{:}\,\mathtt{R}\ \tau_i\,^{i\in I}\ ;\ \mathtt{A}]$ and $\zeta(\chi)[\ell_i{:}\,\mathtt{R}^+\ \tau_i\,^{i\in I}\ ;\ \mathtt{U}]$. One difference is that, in our case, subtyping is defined and permitted field by field rather than all at once. Fisher and Mitchell also studied the relationship between objects and classes in [42]. They use bounded existential quantification to hide some of the structure of the object in the *public* interface. This still allows public methods to be called, while *private* methods become innaccessible. In our calculus, the richer structure of objects permits to use subtyping instead of bounded existential quantification to provide a similar abstraction. This is not suprising, theoretically, since subtyping, as existential quantification, is a lost of type information. However, this is practically a significant difference, since subtyping allows more explicit type information but is less expressive. Another difference is that using the standard record types they had to introduce record sorts to express negative type information. As pointed out in a more recent paper [10], the design of the language of kinds becomes important for modularity. In particular, [10] improves over [42] by changing default kinds from unknown ($\mathtt{U}$ in our setting) to absent ($\mathtt{A}$). Instead, our record types express positive and negative information symmetrically and are viewed as total functions from fields to types, which avoids the somehow *ad hoc* language of sorts.

In a recent paper, Riecke and Stone have circumvented the problem of merging extension with deep and width subtyping by changing the semantics of objects [117]. In fact, their semantics remain in correspondance with the standard semantics of objects in the general case, but the semantics

of extension is changed so that the counter example becomes sound in the new semantics. They distinguish between method update and object extension. Then, a field that is already defined is automatically renamed by extension into an anonymous field that becomes externally inaccessible.

With their semantics, some of our enriched type information would become obsolete for ensuring type soundness, but it might remain useful for compile-time optimizations. Other pieces of information, e.g. virtual types, would remain quite pertinent.

8.8 Conclusion

We have proposed a uniform and flexible method for enriching type systems of object calculi by refining the field structure of object types, so that they carry more precise type information.

Applying our approach to the object calculus of Abadi and Cardelli, we have integrated object extension and depth and width subtyping, with covariant final methods and contra-variant virtual methods, in a type-safe calculus. When sufficient type information is revealed, objects may represent classes. Type information may also be hidden progressively, until objects can be used and interchanged in a traditional fashion.

An important gain is to avoid the encoding of classes as records of pre-methods. Instead, we provide a more uniform, direct approach. Another benefit of this integration is to allow mixed formed of classes and objects. The use of richer object types also increases both safety by capturing more dynamic misbehavior as static type errors and security by allowing more privacy via subtyping. Moreover, our approach subsumes several other unrelated proposals, and it might provide a unified framework for studying or comparing new proposals. Some extensions and variations are clearly possible, provided the operations on objects, their types and the subtyping hierarchy are changed consistently.

More investigation still remains to be done. Adding an equational theory to the calculus, would simplify our primitives, since objects could always be built field by field using object extension only. This might also be a first step towards a better integration of record-based and delegation-based object calculi. In the future, we would also like to study the potential increase of expressiveness that field and row variables could provide. Of course, investigating binary methods remains one of the most important issues.

Classes can be viewed as objects. We hope that an even richer type structure would finally enable to see objects for what they really are —records of functions— in the (yet untyped) self-application interpretation.

Appendix

8.9 Type computation

Lemma 4 (Type computation) *Let τ and τ' be two object types $\zeta(\chi)[\rho]$ and $\zeta(\chi)[\rho']$. Assume that there exists a row ρ'' such that $E, \chi <: \mathtt{T} \vdash \rho <: \rho''$ and for each label ℓ, the pair $(\rho''(\ell), \rho'(\ell))$ is one of the four forms $(\mathtt{A}, \mathtt{P}\ \tau_\ell)$, $(\mathtt{V}\ \tau_\ell, \mathtt{R}\ \tau_\ell)$, $(\mathtt{V}^-\ \tau_\ell, \mathtt{M}^-\ \tau_\ell)$, or (φ, φ). Let $\hat{\tau}$ be $\tau \Leftarrow \tau'$ and $\hat{\rho}$ be $\rho \Leftarrow \rho'$. Then,*

$$E \vdash \hat{\tau} <: \tau' \qquad E \vdash \#\,\hat{\tau} <: \#\,\tau \qquad E \vdash \#\,\hat{\tau} <: \#\,\tau'$$

Moreover, in the three first cases, if $E, \chi <: \mathtt{T} \vdash \rho(\ell) <: \rho'(\ell)$, then $E, \chi <: \mathtt{T} \vdash \rho(\ell) <: \hat{\rho}(\ell)$; otherwise $E, \chi <: \mathtt{T} \vdash \mathtt{P}\ \tau_\ell <: \hat{\rho}(\ell)$.

Proof: Let E' be $E, \chi <: \mathtt{T}$. By rule Sub Obj Invariant it sufficies to check

$$(1)\ E' \vdash \hat{\rho} <: \rho' \qquad (2)\ E' \vdash \#\,\hat{\rho} <: \#\,\rho \qquad (3)\ E' \vdash \#\,\hat{\rho} <: \#\,\rho'$$

$$(4)\ \begin{cases} E' \vdash \rho <: \hat{\rho} & \text{if } \rho'' \text{ is } \rho', \\ \exists \tau'_\ell, E' \vdash \tau_\ell <: \tau'_\ell \wedge E' \vdash \mathtt{P}\ \tau'_\ell <: \hat{\tau}(\ell) & \text{otherwise.} \end{cases}$$

independently for each cell of the table 8.5 and for any of the fourth possible forms (three first forms for (4)).

Case (φ)**:** These cases cannot occur because all hypotheses cannot be met simultaneously.

Case first line: Note that this completely covers the case where (φ'', φ') is $(\mathtt{A}, \mathtt{P}\ \tau_\ell)$. Properties (1) and (3) are immediate since $\hat{\varphi}$ is φ' and (2) is obvious since $\#\,\varphi$ is $\mathtt{U}$. Since $\hat{\varphi}$ is φ', $E' \vdash \varphi <: \hat{\varphi}$ follows from $E' \vdash \varphi <: \varphi'$, and $E' \vdash \varphi' <: \hat{\varphi}$ is always true, hence (4).

Case $E' \vdash \varphi <: \varphi'$**:** In particular, this covers the case where (φ'', φ') is (φ, φ).

Subcase $\hat{\varphi}$ **is** φ**:** Then (1), (2) and (4) are obvious. When $\#\,\varphi'$ is φ', it happens that $\#\,\varphi$ is also φ, thus (3) is true. There are 6 remaining cases in the last two columns:

- In the last column, we must show that $E' \vdash \#\,\varphi <: \mathtt{M}^-\ \tau'$. If φ is $\mathtt{V}^-\ \tau$ or $\mathtt{M}^-\ \tau$, then $E \vdash \tau' <: \tau$, and since $\#\,\varphi$ is $\mathtt{M}^-\ \tau$, then $E' \vdash \#\,\varphi <: \mathtt{V}^-\ \tau'$. Otherwise, φ is either $\mathtt{R}\ \tau$ or $\mathtt{M}\ \tau$, invariant by $\#$ and $E' \vdash \varphi <: \mathtt{M}^-\ \tau$.
- In the preceding column, we must show that $E' \vdash \#\,\varphi <: \mathtt{R}\ \tau'$ (5). Here and since $E' \vdash \varphi <: \varphi'$, φ is one of the form $\mathtt{R}\ \tau$ or $\mathtt{V}\ \tau$, types τ and τ' are equal, and $\#\,\varphi$ is $\mathtt{R}\ \tau$. Hence (5).

Other subcases: given that $E' \vdash \varphi <: \varphi'$, the only remaining subcase is when φ is $\mathtt{M}\ \tau$ and φ' is $\mathtt{V}\ \tau'$. Then τ and τ' are equal and thus so are $\hat{\varphi}$ and φ'. Hence (1) and (3). Clearly, we also have $E' \vdash \mathtt{R}\ \tau <: \mathtt{M}\ \tau$ (2) and $E' \vdash \mathtt{M}\ \tau <: \mathtt{V}\ \tau$ (4).

Case (φ'', φ') **is** $(\mathtt{V}\ \tau_\ell, \mathtt{R}\ \tau_\ell)$**, first line excluded:** Then φ is either $\mathtt{M}\ \tau$ or $\mathtt{V}\ \tau$ with τ and τ_l equal and $\hat{\varphi}$ is $\mathtt{R}\ \tau$. Hence (1) and, since $\hat{\varphi}$ and φ' are here invariant by $\#$, we also have (3). Since both $E' \vdash \mathtt{R}\ \tau <: \mathtt{M}\ \tau$ and $E' \vdash \mathtt{R}\ \tau <: \mathtt{V}\ \tau$, we also have (2). The hypothesis $E' \vdash \varphi <: \mathtt{R}\ \tau_\ell$ never holds. However, $E' \vdash \mathtt{P}\ \tau_l <: \mathtt{R}\ \tau$ holds since τ and τ_l are equal.

Case (φ'', φ') **is** $(\mathtt{V}^-\ \tau_\ell, \mathtt{M}^-\ \tau_\ell)$**, first line excluded:** If φ is either $\mathtt{M}\ \tau$ or $\mathtt{V}\ \tau$ with τ and τ_l equal and we reason as in the previous case. That is τ and τ_l are equal and $\hat{\varphi}$ is $\mathtt{R}\ \tau$. In particular, (2) is unchanged. Since $E' \vdash \mathtt{R}\ \tau_l <: \mathtt{M}^-\ \tau_l$, we also have (1), (3) and $E' \vdash \mathtt{P}\ \tau_l <: \hat{\varphi}$. The hypothesis $E' \vdash \varphi <: \mathtt{M}^-\ \tau_\ell$ holds when φ is $\mathtt{M}\ \tau$, and then (4) holds since $\hat{\varphi}$ is φ.

Otherwise, φ is either $\mathtt{M}^-\ \tau$ or $\mathtt{V}^-\ \tau$ with $E' \vdash \tau_\ell <: \tau$ and $\hat{\varphi}$ is $\mathtt{M}^-\ \tau$. Since $E' \vdash \mathtt{M}^-\ \tau <: \mathtt{M}^-\ \tau_\ell$, i.e. $E' \vdash \hat{\varphi} <: \varphi'$, we have (1). Since $\mathtt{M}^-$, i.e. both sides, are invariant by $\#$, we also have (3). Since both $\#\,\varphi$ is equal to $\hat{\varphi}$, which is invariant by $\#$, we also have (2). The inequality $E' \vdash \varphi <: \varphi'$ holds when φ is $\mathtt{M}^-\ \tau$, i.e. $\hat{\varphi}$, and then $E' \vdash \varphi <: \hat{\varphi}$ trivially holds. Otherwise, (4) holds taking τ for τ'_ℓ.

■

8.10 Subject reduction

Theorem 1 (Subject Reduction) *Typings are preserved by reduction. If* $E \vdash a : \tau_a$ *and* $a \longrightarrow a'$ *then* $E \vdash a' : \tau_a$.

Proof: By induction on the size of a and cases on the reduction.

Case Red Select: The expression a is of the form $v.\ell_j$ where v is $\zeta(\chi,\tau)[\ell_i = \varsigma(x)a_i{}^{i\in I}]$ and j is in I. It reduces to $a_j\{\tau/\chi\}\{v/x\}$. The derivation of $E \vdash a : \tau_a$ ends with a subsumption rule preceded by rule Expr Select. Thus, there exists types τ' and τ''_j such that

$$E \vdash v : \tau' \qquad E \vdash \tau' <: \zeta(\chi)[\ell_j{:}\,\mathtt{R}^+\,\tau''_j\,;\mathtt{U}] \ (8.1) \qquad E \vdash \tau''_j\{\tau'/\chi\} <: \tau_a \ (8.2)$$

The derivation $E \vdash v : \tau'$ itself ends with a subsumption rule preceded by rule Expr Object. Thus, τ is of the form $\zeta(\chi)[\rho]$ and there exist $\tau_i{}^{i\in I}$ such that

$$E, \chi <: \#\,\tau, x_i : \chi \vdash a_i : \tau_i, \ \ \forall i \in I \ (8.3) \qquad E, \chi <: \#\,\tau \vdash (\ell_i : \mathtt{P}\ \tau_i{}^{i\in I}; \mathtt{A}) <: \rho \ (8.4)$$

$$E \vdash \tau <: \tau' \ (8.5)$$

By transitivity between (8.5) and (8.1), we have $E \vdash \tau <: \zeta(\chi)[\ell_j{:}\,\mathtt{R}^+\,\tau''_j\,; U]$ (8.6). By structural subtyping (lemma 39), and transitivity with (8.4), we have, in particular, $E, \chi <: \mathtt{T} \vdash \mathtt{P}\ \tau_j <: \mathtt{R}^+\,\tau''_j$.

By structural subtyping (lemma 39), $E, \chi{<:}\mathtt{T} \vdash \tau_j{<:}\tau''_j$. Thus, by subsumption applied to (8.3), $E, \chi <: \#\,\tau, x_i : \chi \vdash a_i : \tau''_j$ (8.7). The judgment (8.6) also implies that $E \vdash \tau <: \zeta(\chi)[\mathtt{U}]$, and by lemma 41 we have $E \vdash \tau <: \#\,\tau$. Therefore, applying substitution (lemma 38) to (8.7), we have $E, x_i : \tau \vdash a_i\{\tau/\chi\} : \tau''_j\{\tau/\chi\}$. Since $E \vdash v : \tau$, by substitution again, we have $E \vdash a_i\{\tau/\chi\}\{v/x\} : \tau''_j\{\tau/\chi\}$ (8.8). Since τ''_j is covariant, it follows from (8.5) that $E \vdash \tau''_j\{\tau/\chi\} <: \tau''_j\{\tau'/\chi\}$. By transitivity with (8.2), $E \vdash \tau''_j\{\tau/\chi\} <: \tau_a$ (8.9). We conclude using subsumption applied to (8.8) with (8.9).

Case Red Extend: The expression a is of the form $v.\ell \Leftarrow \zeta(\chi,\tau')\varsigma(x_0)a_0$ where v is $\zeta(\chi,\tau)[\ell_i = \varsigma(x_i)a_i{}^{i\in I}]$ and ℓ is not one of the ℓ_i's. It reduces to $\zeta(\chi,\hat\tau)[\ell = \varsigma(x_0)a_0\ ; \ell_i = \varsigma(x_i)a_i{}^{i\in I}]$ where $\hat\tau$ is $\tau \Leftarrow \tau'$.

Let τ and τ' be $\zeta(\chi)[\rho]$ and $\zeta(\chi)[\rho']$ (this is not restrictive) and $\hat\rho$ be $\rho \Leftarrow \rho'$. A derivation of $v.\ell_j \Leftarrow \zeta(\chi,\tau')\varsigma(x)a$ ends with a subsumption rule preceded by rule Expr Extend. Thus, τ' verifies:

$$(\varphi_0, \rho'(\ell)) \in \{(\mathtt{A}, \mathtt{P}\ \tau_\ell), (\mathtt{V}\ \tau_\ell, \mathtt{R}\ \tau_\ell), (\mathtt{V}^-\,\tau_\ell, \mathtt{M}^-\,\tau_\ell)\} \ (8.1) \qquad E \vdash v : \zeta(\chi)[\ell{:}\,\varphi_0\ ;\ \rho'] \ (8.2)$$

$$E, \chi <: \#\,\tau', x_0 : \chi \vdash a_0 : \tau_\ell \ (8.3) \qquad E \vdash \tau' <: \tau_a \ (8.4)$$

A derivation of (8.2) ends with subsumption preceded by rule Expr Object. Thus, τ verifies:

$$E, \chi <: \#\,\tau, x_i : \chi \vdash a_i : \tau_i, \ \forall i \in I \ (8.5) \qquad E, \chi <: \#\,\tau \vdash (\ell_i : \mathtt{P}\ \tau_i{}^{i\in I}; \mathtt{A}) <: \rho \ (8.6)$$

$$E \vdash \tau <: \zeta(\chi)[\ell{:}\,\varphi_0\ ;\ \rho'] \ (8.7)$$

From (8.7), by structural subtyping, we have $E, \chi{<:}\mathtt{T} \vdash \rho{<:}(\ell : \varphi_0; \rho')$. Thus, $E, \chi{<:}\mathtt{T} \vdash \rho\backslash\ell{<:}\rho'\backslash\ell$. This, together with (8.1) meets the hypotheses of lemma 40. Therefore,

$$E \vdash \hat\tau <: \tau' \ (8.8) \qquad E \vdash \#\,\hat\tau <: \#\,\tau \ (8.9) \qquad E \vdash \#\,\hat\tau <: \#\,\tau' \ (8.10)$$

Moreover, for some τ'_ℓ:
$E, \chi <: \#\,\tau \vdash \rho\backslash\ell <: \hat\rho\backslash\ell$ (8.11) $\qquad E, \chi <: \#\,\tau \vdash \tau_\ell <: \hat\tau'_\ell$ (8.12) $\qquad E, \chi <: \#\,\tau \vdash \mathtt{P}\ \tau'_\ell <: \hat\rho(\ell)$ (8.13)

Combining (8.6), (8.11), and (8.13) we have $E, \chi <: \#\,\tau \vdash (\ell : \mathtt{P}\ \tau'_\ell; \ell_i : \mathtt{P}\ \tau_i{}^{i\in I}; \mathtt{A}) <: \hat\rho$. By bound weakening, since (8.9), we get $E, \chi <: \#\,\hat\tau \vdash (\ell : \mathtt{P}\ \tau'_\ell; \ell_i : \mathtt{P}\ \tau_i{}^{i\in I}; \mathtt{A}) <: \hat\rho$ (8.14). Combining (8.3) with (8.12), we have $E, \chi <: \#\,\tau', x_0 : \chi \vdash a_0 : \tau'_\ell$ (8.15). By substitution lemma applied to (8.15) and (8.5), with (8.9) we have

$$E, \chi <: \#\,\hat\tau, x_0 : \chi \vdash a_0 : \tau'_\ell \qquad E, \chi <: \#\,\hat\tau, x_i : \chi \vdash a_i : \tau_i, \ \forall i \in I$$

Combining with (8.14), we have $E \vdash a' : \hat\tau$. By subsumption applied with (8.8) and (8.4), we finally have $E \vdash a' : \tau_a$.

Case Red Update: We reuse the same notations. The difference is that ℓ is now one the ℓ_j for j in I. The expression a is of the form $v.\ell \Leftarrow \zeta(\chi,\tau')\varsigma(x)a$ where v is $\zeta(\chi,\tau)[\ell_i = \varsigma(x_i)a_i{}^{i\in I}]$. It reduces to $\zeta(\chi,\hat{\tau})[\ell_j = \varsigma(x)a\ ;\ \ell_i = \varsigma(x_i)a_i{}^{i\in I-j}]$.

We distinguish two subcases according to form of the typing derivation for a.

Subcase Expr Extend: This case is similar to the case for extension. The only differences in the proof if that here, from (8.6), (8.9) and (8.13), we have $E,\chi <: \#\hat{\tau} \vdash (\ell : \mathtt{P}\ \tau_\ell; \ell_i : \mathtt{P}\ \tau_i{}^{i\in I-j}; \mathtt{A}) <: \hat{\rho}$ instead of (8.14).

Subcase Expr Update: A derivation of $v.\ell \Leftarrow \zeta(\chi,\tau')\varsigma(x_0)a_0$ ends with a subsumption rule preceded by rule Expr Update. Thus, τ' verifies:

$$E \vdash v : \tau' \ (8.1) \qquad E \vdash \tau' <: \zeta(\chi)[\ell{:}\,\mathtt{R}\ \tau_\ell\ ;\ \rho_0] \ (8.2) \qquad E,\chi <: \#\tau', x_0 : \chi \vdash a_0 : \tau_\ell \ (8.3)$$

$$E \vdash \tau' <: \tau_a \ (8.4)$$

A derivation of (8.1) ends with subsumption preceded by rule Object. Thus, τ verifies:

$$E,\chi <: \#\tau, x_i : \chi \vdash a_i : \tau_i,\ \forall i \in I \ (8.5) \qquad E,\chi <: \#\tau \vdash (\ell_i : \mathtt{P}\ \tau_i{}^{i\in I}; \mathtt{A}) <: \rho \ (8.6)$$

$$E \vdash \tau <: \tau' \ (8.7)$$

From (8.7), by structural subtyping, we have $E,\chi <: \mathtt{T} \vdash \rho <: \rho'$, which enables to apply lemma 40; we get

$$E \vdash \hat{\tau} <: \tau' \ (8.8) \qquad E \vdash \#\hat{\tau} <: \#\tau (8.9) \qquad E \vdash \#\hat{\tau} <: \#\tau' (8.10) \qquad E,\chi <: \#\tau \vdash \rho <: \hat{\rho} \ (8.11)$$

Combining (8.11) with (8.6), we have $E,\chi <: \#\tau \vdash (\ell_i : \mathtt{P}\ \tau_i{}^{i\in I}; \mathtt{A}) <: \hat{\rho}$. By bound weakening, since (8.9), we have $E,\chi <: \#\hat{\tau} \vdash (\ell_i : \mathtt{P}\ \tau_i{}^{i\in I}; \mathtt{A}) <: \hat{\rho}$ (8.12). By structural subtyping (lemma 39) applied to (8.2), we have $E,\chi <: \mathtt{T} \vdash \rho'(\ell) <: \mathtt{R}\ \tau_\ell$. By bound weakening with (8.10), we have $E,\chi <: \#\hat{\tau} \vdash \rho'(\ell) <: \mathtt{R}\ \tau_\ell$. By transitivity with (8.8) after applying structural subtyping, we get $E,\chi <: \#\hat{\tau} \vdash \hat{\rho}(\ell) <: \mathtt{R}\ \tau_\ell$. By structural subtyping, we must have $E,\chi <: \#\hat{\tau} \vdash \mathtt{P}\ \tau_\ell <: \hat{\rho}(\ell)$. Combining this with (8.12), we have $E,\chi <: \#\hat{\tau} \vdash (\ell : \mathtt{P}\ \tau_\ell; \ell_i : \mathtt{P}\ \tau_i{}^{i\in I-j}; \mathtt{A}) <: \hat{\rho}$ (8.13).

By substitution lemma applied to (8.3) and (8.5), we have

$$E,\chi <: \#\hat{\tau}, x : \chi \vdash a : \tau_\ell \qquad E,\chi <: \#\hat{\tau}, x_i : \chi \vdash a_i : \tau_i,\ \forall i \in I$$

Combining with (8.13), we have $E \vdash a' : \hat{\tau}$. By subsumption applied with (8.8) and (8.4), we finally have $E \vdash a' : \tau_a$.

Case Context: Trivial using the induction hypothesis.

■

Chapitre 9

Conclusions

Nous avons exploré un des chemins possibles des enregistrements aux objets. Le remplacement des enregistrements déclarés par des enregistrements polymorphes et extensibles, dans le langage ML, puis l'ajout d'une opération de concaténation nous a permis de voir les objets comme des enregistrements de fonctions et les classes comme des fonctions sur ces enregistrements. Le retour à des objets et des classes primitifs a facilité la mise en œuvre d'un mécanisme d'abréviation automatique corrigeant les problèmes d'interface liés à la structure ouverte des types d'objets. Cela a abouti au langage Objective ML. C'est le résultat concret le plus important de notre travail. Les derniers chapitres étudient des améliorations du langage Objective ML. Tout en apportant des réponses précises, ils soulèvent aussi de nouvelles questions. L'ajout du polymorphisme de première classe est-il un premier pas vers l'abandon de la synthèse des types en ML et à terme l'abandon de ML ? Ou, au contraire, est-ce une nouvelle porte qui s'ouvre vers des systèmes de types plus expressifs qui conserveraient l'essence donc la pérennité du langage ML ?

L'ensemble des travaux présentés s'appuient, autant dans les motivations initiales que dans la formulation finale sur la correspondance étroite entre objets et enregistrements. Le dernier chapitre va au delà en identifiant objets et classes. La compréhension de la programmation des langages à objets ne doit pas se limiter à la compréhension des objets, et la prise en compte de la notion de classe dès le départ est un atout.

Le langage Objective ML, n'est sans doute pas une proposition finale, et il devrait pouvoir être simplifié davantage. Pourtant, un peu comme s'il avait atteint un point d'équilibre minimal, plusieurs tentatives en ce sens ont échoué : elles ont entraîné une perte intolérable d'expressivité, ou inversement, une complication démesurée de la sémantique statique ou dynamique. Mais nous restons optimistes quant à l'existence d'une présentation d'Objective ML plus légère et encore plus expressive ! Cela pourrait passer par une meilleure identification des objets et des classes, ou bien par une généralisation de l'opération d'extension dans l'esprit de [117].

Résultats théoriques et pratiques

Bien que le but de notre étude soit l'extension de ML avec des objets et des classes, notre parcours est jalonné de résultats auxiliaires.

L'extension de ML avec des enregistrements polymorphes est une étape essentielle sur le plan théorique. Elle est le point d'articulation et le fondement de tous les travaux présentés ici. C'est en fait la mise en forme des variables de rangée dont le concept avait été introduit dans [119], et de l'ensemble des mécanismes qui permettent de les intégrer à ML tout en conservant la synthèse des types. Une idée capitale est l'adjonction des annotations de présence au niveau des types qui

permet de traiter les types enregistrements comme des fonctions totales plutôt que des fonctions partielles sur les étiquettes. Cette décomposition augmente à la fois l'expressivité et l'uniformité donc également la simplicité des types enregistrements. Mais le principe est général, et nous l'avons appliqué avec avantage aux types objets dans le chapitre 8 dans un contexte explicitement typé.

Bien que les enregistrements polymorphes extensibles ne soient pas intégrés dans le langage Objective Caml (parce que les objets les rendent moins nécessaires, voire un peu superflus), cette extension a plusieurs applications concrètes. Elle a été reprise (dans un cas particulier) pour le typage des objets. Elle est aussi utilisée, de façon plus essentielle pour le typage des variantes avec sous-typage [97].

L'enrichissement des types enregistrements proposé dans le chapitre 3 est utilisé de façon un peu cachée mais essentielle dans le chapitre 5 pour définir les types des objets et des messages. Cette extension permet aussi de typer des enregistrements avec des valeurs par défaut, rapprochant ainsi les enregistrements de la forme duale des variantes (les variantes ayant naturellement un cas par défaut).

Le typage de la concaténation des enregistrements est une technique autant qu'une proposition qui consiste à simuler la composition de fonctions d'extensions associées aux enregistrements. Cette méthode est utilisée à nouveau dans le chapitre 5 pour modéliser directement l'héritage multiple. Cela revient à voir les classes comme des *mixins* et ainsi remplacer l'héritage multiple par la composition des mixins. Il manquait au concept des mixins, connu mais trop peu considéré, un lien plus clair avec l'héritage multiple et la concaténation des enregistrements que nous avons établi et valorisé dans ML-ART. Dans le langage Objective ML, nous avons préféré revenir à une opération d'héritage multiple parce qu'elle était plus standard, mais aussi parce qu'elle conduit à des types plus lisibles, ce qui est très important dans un contexte de synthèse des types. Mais la notion de mixin reste avantageuse. Elle est de nouveau valorisée dans [43]. Nous pourrions aussi la réintroduire dans une future version d'Objective ML, maintenant que la synthèse des types et leur mécanisme d'abréviation sont bien compris.

Objective ML est un langage avec des objets primitifs et une synthèse complète des types. Mais il ne doit pas être confiné à demeurer une extension de ML. C'est aussi une approche du typage des objets qui repose de façon essentielle sur le polymorphisme des variables de rangée qui malgré les contraintes (ou grâce à elles) imposées par la volonté de préserver la synthèse de types s'avère extrêmement expressive et possède bien des atouts que bien d'autres propositions, typés explicitement, peuvent lui envier. De nombreuses constructions, réputées difficiles, comme les opérations réflexives y compris les méthodes binaires s'y expriment simplement.

Ces propriétés résultent de l'usage du polymorphisme des variables de rangée plutôt que du polymorphisme de sous-typage. En ce sens nous partageons l'avis de Kim Bruce que le sous-typage n'est pas la bonne notion pour la programmation à objets. Dans le langage PolyToil [18] il introduit comme alternative une notion de filtrage (*matching*) sur les types. Or, cette nouvelle opération, traitée de façon un peu *ad hoc*, correspond par essence au polymorphisme des variables de rangée. Objective ML et PolyToil partagent de nombreuses qualités, notamment la facilité à traiter le type de self et les méthodes binaires.

Toutefois, nous pensons que l'approche par variables de rangée est supérieure au filtrage parce que le polymorphisme est un concept simple, bien connu, et déjà présent dans les langages considérés. De plus le polymorphisme des variables de rangée est une forme spécialisée d'une notion plus générale de destructeur de type introduite récemment dans [53]. Le polymorphisme des variables de rangée permet aussi d'exprimer l'extension des enregistrements. Toutefois, il ne semble pas être suffisant pour coder l'extension des objets présentée dans le chapitre 8.

Objective ML est donc aussi la démonstration que le polymorphisme des variables de rangée

remplace avantageusement le sous-typage mais aussi la quantification filtrée. Sans être tout à fait aussi général, il s'arrête là où l'un ou l'autre deviennent simultanément plus compliqués et moins utiles.

L'ajout du polymorphisme de première classe répond au besoin réel de méthodes polymorphes dans les classes paramétriques. C'est aussi l'élimination d'un point faible d'Objective ML, et il a valeur de symbole. Mais au delà de l'application aux objets c'est aussi une réponse à un besoin latent de petites doses d'ordre supérieur en ML. Ce problème avait reçu trop peu d'attention et les solutions de principe proposée n'étaient pas très satisfaisantes en pratique. Ce résultat est aussi une ouverture de ML vers un langage mixte avec typage implicite et explicite et synthèse partielle des types.

L'identification des objets et des classes est pour l'instant un travail plutôt théorique qui simplifie plusieurs propositions existantes. Il corrige aussi un défaut de nombreuses études qui tendent à reléguer les classes au second plan, et souvent les traitent par des codages. Mais son intérêt est surtout dans les prolongements auxquels il invite.

Les prolongements

Objective ML est un accomplissement, mais aussi une étape. Ses qualités ne doivent pas cacher pour autant les améliorations possibles qui restent à faire. Objective ML est aujourd'hui la seule version de ML distribuée avec des objets. Au delà des extensions envisageables, il faut continuer à rechercher d'autres ajouts, directement liés aux objets, ou d'autres alternatives à la programmation avec objets. Comme nous l'avons rappelé en introduction, la recherche de constructions plus expressives, plus simples, sûres et efficaces est sans fin. Elle ne peut être abandonnée. Nous décrivons ci-dessous quelques directions de recherche pour étendre les travaux présentés ici.

Types d'ordre supérieur

L'ajout de types d'ordre supérieur décrit dans le chapitre 7 à été motivé avant tout par le besoin de méthodes polymorphes dans les objets et cela reste sa plus importante application. Toutefois, cette approche qui généralise la proposition plus facile faite dans le chapitre 5, doit être replacée dans un cadre plus large de la quête d'un système intermédiaire entre ML et le système F. Cette idée n'est pas nouvelle, mais elle se trouve réactivée aujourd'hui par le besoin plus pressant d'un système de types plus riche comme celui du système $F^{\omega}_{<:}$, et la volonté de préserver la synthèse des types pour continuer à être, sinon prétendre, de la famille ML.

Notre proposition s'applique parfaitement aux méthodes polymorphes. Si l'on se restreint au noyau ML, la solution proposée est convenable, mais on ressent le besoin d'aller au delà et de rendre encore plus implicite l'élimination du polymorphisme. Nous avons déjà commencé à explorer cette direction, mais la solution n'est pas pour l'instant satisfaisante. Il serait aussi intéressant d'étudier l'interaction de cette proposition avec une forme semi-implicite de sous-typage.

Sous-typage

Nous avons mentionné à plusieurs reprises le sous-typage tout en donnant l'impression de l'avoir ignoré ou critiqué. Tant l'absence de sous-typage est une faiblesse de ML, tant sa relégation au second plan est un point fort d'Objective ML. À notre connaissance, Ocaml, PolyToil et ses descendants cités ci-dessus sont parmi les seuls langages à ne pas s'appuyer sur un mécanisme de sous-typage. Simultanément, ils traitent tous deux facilement les méthodes binaires.

Nous n'avons pas pour autant négligé le sous-typage. Il permet d'insérer des coercions entre certaines valeurs ayant une même représentation mais des types différents. Il est présent, explicitement déclaré dans Objective ML, où il s'avère très utile, mais relativement peu souvent.

Le sous-typage avec inférence de types a été étudié pour ML et pour une extension de ML avec des objets dans [40]. Toutefois les problèmes d'interface déjà difficiles en ML-ART se trouvent décuplés par la présence du sous-typage. Bien qu'il ne soit pas prépondérant, l'usage du sous-typage pourrait être facilité en Objective ML s'il était synthétisé. Les travaux de François Pottier [97] permettent d'envisager maintenant l'étude d'une version d'Objective ML avec synthèse des sous-types. Cependant, il restera —et c'est le plus difficile— à ajuster les techniques d'abréviation automatique des types en présence de sous-typage implicite.

Typage des objets et abstraction

Le typage des objets dans Objective ML est très satisfaisant ; toutefois, son expressivité, la robustesse et la simplicité de sa mise en œuvre et sa facilité d'utilisation récemment améliorée, devraient maintenant laisser apparaître les plus petits défauts.

Le problème le plus évident aujourd'hui est de ne pas pouvoir cacher les méthodes a posteriori, ce qui nuit à la modularité : une classe écrite en librairie révélera toutes les fonctionnalités qui ne compromettent ni la sécurité ni l'abstraction. Un utilisateur qui trouve cette classe en librairie, mais avec une interface plus restreinte, ne peut parfois pas l'utiliser parce qu'une partie non désirée de l'interface ne peut être cachée.

Ce problème sérieux trouve son origine aux sources des objets dans le mécanisme de liaison tardive. Par défaut, l'oubli de structure est une opération implicite qui ne change pas la représentation des objets. Le mécanisme de liaison tardive rend certaines de ces opérations dangereuses. Les outils développés dans le chapitre 8 permettent de tracer les opérations d'oubli, et autorisent les opérations sûres tout en détectant celles qui sont dangereuses. Mais cette approche n'est pas toujours suffisante. Il est des cas où l'on veut effectivement cacher des méthodes, et redéfinir d'autres méthodes du même nom, mais indépendantes. Il faut pour cela un moyen d'interrompre la liaison tardive. Une proposition récente [117] enrichit le mécanisme de liaison tardive de façon à pouvoir autoriser l'oubli de méthodes sans restriction. Il serait intéressant de voir comment cette nouvelle construction peut se combiner avec la solution orthogonale proposée dans 8. Une autre direction est de reprendre cette proposition dans le cadre d'Objective ML et de mieux comprendre son adaptation au polymorphisme des variables de rangée et son interaction avec les méthodes binaires.

Il est remarquable que la proposition précédente [117] ou une variante [43] qui étend la sémantique des objets soit le résultat d'un problème de typage. Le typage des objets qui jusqu'à présent se contentait de "rattraper" son retard par rapport aux langages à objets non typés est maintenant bien mature et propose des constructions nouvelles, utiles également dans un contexte non typé.

Modules et objets

Le langage ML possède à la fois un mécanisme de modules sophistiqué et un langage d'objets performant. Victime de sa réussite multiple, se pose maintenant un nouveau problème que l'on ne doit pas sous-estimer. Les notions de modules et d'objets ont été motivées par des besoins similaires de structuration de la programmation, d'abstraction, de partage et de réutilisation de code. L'une et l'autre répondent aujourd'hui à ces besoins avec succès, mais aussi de façon très différente [118].

Alors que modules et objets se rejoignent dans leur accomplissement, ils diffèrent dans leur mise en œuvre. Pire, ils induisent chacun des styles de programmation divergents et peu compatibles.

Ainsi, pour des raisons historiques les librairies écrites dans un style modulaire incitent peu à l'utilisation des objets. Inversement, une application écrite en style objet nécessite souvent de réécrire une partie des librairies dans un style compatible.

Il est essentiel de rapprocher les deux notions et de donner à l'utilisateur une vision uniforme de la programmation incrémentale et modulaire.

Co-objets

Il existe traditionnellement deux approches de la programmation avec objets. Celle que nous avons étudiée est dite à enregistrements parce que les objets peuvent être considérés comme des enregistrements portant leurs méthodes, même si une implantation particulière peut les représenter différemment. L'autre approche, dite à surcharge, considère que les objets ne portent que leurs variables d'instance. Les méthodes sont alors des fonctions externes, globales, surchargées. Alors que le typage des objets dans l'approche à enregistrements est aujourd'hui très bien compris, le typage des méthodes comme fonctions surchargées est encore fort peu satisfaisant. Les mécanismes de surcharge, déjà difficiles, ont trouvé quelques solutions [28, 29, 14], mais l'application ultérieure aux objets passe encore par une analyse ou une transformation globale.

Cette difficulté de typage cache probablement une difficulté sémantique plus profonde. Pourtant, l'approche à surcharge est par certains aspects plus intuitive. Il faut aussi reconnaître que cette approche a concentré beaucoup moins d'effort. Fort de notre expérience et munis de bons instruments, il serait intéressant de rechercher un autre chemin menant des sommes aux objets.

Objets distribués

La distribution prend une importance croissante dans tous les systèmes donc aussi dans la plupart des langages de programmation.

Le transfert des techniques de typage développées pour les objets en Objective ML au calcul joint à été entrepris et ne pose pas de problème particulier. Cependant, il est intéressant de le mener à bien, en espérant surtout profiter du contexte distribué pour trouver de nouvelles contraintes et d'autres exigences. Cela risque de nous amener à nous orienter vers des constructions légèrement différentes, et ne peut à terme qu'enrichir notre compréhension des objets.

Épilogue

S'il fallait retracer demain la route des objets, le point d'arrivée serait sensiblement le même, mais la ligne serait droite, sans virage, courte et directe. Il est toujours intrigant de regarder derrière soi et de s'apercevoir de la faible distance qui sépare le point d'arrivée du point de départ. Mais le fait que le trajet puisse être parcouru aujourd'hui en un temps plus court est le signe que notre compréhension des objets est devenue mature et que les nouvelles constructions ont été rattachées à des ancrages solides et connus de tous.

Alors que la plupart des recherches sur les objets d'hier s'attachaient à comprendre et formaliser rigoureusement les nombreux concepts introduits pour la plupart il y a plus de deux décennies, les travaux en cours proposent de nouvelles opérations plus puissantes, mais aussi souvent simplificatrices et unificatrices. Tantôt les notions d'objets et de classes se décomposent en atomes plus élémentaires que sont les enregistrements, les fonctions et l'abstraction, tantôt elles deviennent primitives, fusionnent et englobent tous ces aspects à la fois.

Notre sentiment de bien comprendre les objets ne peut pas pour autant nous empêcher de rêver à des objets meilleurs. Comme nous l'avons rappelé au début de ce mémoire, la recherche de nouvelles structures de programmation est une histoire sans fin. Au fur et à mesure que certains concepts s'éclaircissent, que des aspérités disparaissent, notre toucher devient plus sensible et de plus petites rugosités surgissent.

Les concepts d'objets et de modules, apparus simultanément par fission de l'idée de programmation structurée et incrémentale et de réutilisabilité, composés avec la même matière, sont parfois aux antipodes l'un de l'autre. Il reste à ouvrir la voie, encore presque vierge qui les reliera en attente de leur inévitable fusion.

Bibliographie

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects : Untyped and first-order systems. In *Theoretical Aspects of Computer Software*, pages 296–320. Springer-Verlag, April 1994.

[2] Martín Abadi and Luca Cardelli. A theory of primitive objects : Second-order systems. *Science of Computer Programming*, 25(2-3) :81–116, December 1995. Preliminary version appeared in D. Sanella, editor, Proceedings of European Symposium on Programming, pages 1-24. Springer-Verlag, April 1994.

[3] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer, 1996.

[4] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM press, 1993.

[5] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 104–118, Orlando, FL, January 1991. Also available as DEC Systems Research Center Research Report number 62, August 1990.

[6] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *Transactions on Programming Languages and Systems. ACM*, 15(4) :575–631, 1993.

[7] Bernard Berthomieu. Ccs programming in an ml framework : An account of lcs. In F. Nielson, editor, *ML with Concurrency : Design, Analysis, Implementation, and Application*, Monographs in Computer Science, chapter 4. Springer-Verlag, January 1997.

[8] Bernard Berthomieu and Camille le Moniès de Sagazan. A calculus of tagged types, with applications to process languages. In *TAPSOFT Workshop on Types for Program Analysis, Aarhus, Denmark*, May 1995. (DAIMI report PB-493, University of Aarhus).

[9] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping Constraints for Incomplete Objects. In *Proceedings of TAPSOFT-CAAP-97, International Joint Conference on the Theory and Practice of Software Development*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[10] V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In *Informal Proceedings of the FOOL 5 workshop on Foundations of Object Oriented Programming*, Sans Diego, CA, January 1998.

[11] V. Bono and L. Liquori. A subtyping for the fisher-honsell-mitchell lambda calculus of object. In *Proc. of CSL-94, International Conference of Computer Science Logic*, volume 933 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1995.

[12] Viviana Bono and Michele Bugliesi. A lambda calculus of incomplete objects. In *Proceedings of Mathematical Foundations of Computer Science(MFCS)*, number 1113 in Lecture Notes in Computer Science, pages 218–229, 1996.

[13] Viviana Bono and Michele Bugliesi. Matching constraints for the lambda calculus of objects. In *Proceedings of (MFCS)*, 1997.

[14] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages*, pages 302–315, July 1997.

[15] Kim B. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[16] Kim B. Bruce. Typing in object-oriented languages : Achieving expressibility and safety. Revised version to appear in Computing Surveys, November 1995.

[17] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Valery Trifonov) the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3) :221–242, 1996.

[18] Kim B. Bruce, Angela Schuett, and Robert van Gent. Polytoil : A type-safe polymorphic object-oriented language. In *ECOOP*, number 952 in LNCS, pages 27–51. Springer Verlag, 1995.

[19] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[20] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–68. Springer Verlag, 1984. Also in Information and Computation, 1988.

[21] Luca Cardelli. Amber. In *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 21–47. Spinger Verlag, 1986. Proceedings of the 13th Summer School of the LITP.

[22] Luca Cardelli. Typefull programming. In *IFIP advanced seminar on Formal Methods in Programming Langage Semantics*, Lecture Notes in Computer Science. Springer Verlag, 1989.

[23] Luca Cardelli. An implementation of FSub. Research Report 97, Digital Equipment Corporation Systems Research Center, 1993.

[24] Luca Cardelli. Extensible records in a pure calculus of subtyping. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1994.

[25] Luca Cardelli. Extensible records in a pure calculus of subtyping. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*, pages 373–425. MIT Press, 1994.

[26] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4) :417–458, October 1991. Preliminary version in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC SRC Research Report 55, Feb. 1990.

[27] Luca Cardelli and John C. Mitchell. Operations on records. In *Fifth International Conference on Mathematical Foundations of Programming Semantics*, 1989.

[28] G. Castagna. A meta-language for typed object-oriented languages. *Theoretical Computer Science*, 151(2) :297–352, November 1995. Extended abstract in the Proceedings of the *13th*

Conference on the Foundations of Software Technology and Theoretical Computer Science ; Lecture Notes in Computer Science, number 761, December 1993.

[29] Craig Chambers and Gary Leavens. BeCecil, a core object-oriented lnaguage with block structure and multimethods : Semantics and typing. Presented at the FOOL'4 workshop, January 1997.

[30] Lucky Chillan. *Une extension de ML avec des aspects orientés objets.* Thèse de doctorat, Université de Paris 7, Place Jussieu, Paris, France, 1993. Forthcoming.

[31] M. Coppo and P. Giannini. A complete type inference algorithm for simple intersection types. In *Proceedings of the seventeenth Colloquium on trees in Algebra and Programming (CAAP'92)*, volume 581 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[32] Mario Coppo. An extended polymorphic type system for applicative languages. In *MFCS '80*, volume 88 of *Lecture Notes in Computer Science*, pages 194–204. Springer Verlag, 1980.

[33] Guy Cousineau and Gérard Huet. *The CAML Primer*. INRIA-Rocquencourt, France, 1989.

[34] Pavel Curtis. *Constrained Quantification in Polymorphic Type Analysis*. PhD thesis, Cornell, 1987.

[35] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Nineteenth Annual Symposium on Principles Of Programming Langages*, pages 207–212, 1982.

[36] Roberto Di Cosmo. Type isomorphisms in a type-assignment framework. In *19th Symposium on Principles of Programming Languages*, pages 200–210. ACM Press, 1992.

[37] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Higher-order unification via explicit substitutions : the case of higher-order patterns. In M. Maher, editor, *Joint international conference and symposium on logic programming*, pages 259–273, 1996.

[38] Dominic Duggan. Polymorphic methods with self types for ML-like languages. Technical report CS-95-03,, University of Waterloo, 1995.

[39] Dominic Duggan. Polymorphic methods with self types for ML-like languages. Technical report cs-95-03, University of Waterloo, 1995.

[40] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA*, 1995.

[41] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*, 1995.

[42] K. Fisher and J. C. Mitchell. On the relationship between classes, objects and data abstraction. *Theoretical And Practice of Objects Systems*, To appear, 1998. A preliminary version appeared in the proceedings of the International Summer School on Mathematics of Program Construction, Marktoberdorf, Germany, Springer LNCS, 1997.

[43] Matthew Flatt, Shiriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *25th Symposium on Principles of Programming Languages*, pages 171–183, Sans Diego, CA, jan 1998.

[44] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, pages 94–114. Springer Verlag, 1988.

[45] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference : Closing the theory-practice gap. In *TAPSOFT'89*, 1989.

[46] Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. In *International Symposium on Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 20–46. Springer, September 1997.

[47] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, June 1995.

[48] Robert Harper, Robin Milner, and Mads Tofte. *The definition of Standard ML.* The MIT Press, 1991.

[49] Robert W. Harper and Benjamin C. Pierce. Extensible records without subsumption. Technical Report CMU-CS-90-102, Carnegie Mellon University, Pittsburg, Pensylvania, February 1990.

[50] Robert W. Harper and Benjamin C. Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando FL*, pages 131–142. ACM, January 1991. Extended version available as Carnegie Mellon Technical Report CMU-CS-90-157.

[51] Andreas V. Hense. An O'small interpreter based on denotational semantics. Technical Report A 07/91, Universitıat des Saarlandes, Fachbereich 14, 1991.

[52] Andreas V. Hense. Wrapper semantics of an object oriented programming language with state. *Theoretical Aspects of Computer Science*, Lecture notes in Computer Science(526), September 1991.

[53] Martin Hofmann and Benjamin Pierce. Type destructors. In *Informal proceedings of the FOOL'5 workshop*, 1998. Avalaible electronically at http ://pauillac.inria.fr/ remy/fool/.

[54] Gérard Huet. *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots, \omega$. Thèse de doctorat d'état, Université Paris 7, 1976.

[55] Lalita A. Jategaonkar. ML with extended pattern matching and subtypes. Master's thesis, MIT, 545 Technology Square, Cambridge, MA 02139, August 89.

[56] Lalita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

[57] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras : a rule-based survey of unification. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. MIT-Press, Cambridge (MA, USA), 1991.

[58] A. J. Kennedy. Type inference and equational theories. Technical report LIX/RR/96/09, Ecole Polytechnique, LIX, Ecole Polytechnique, 91128 Palaiseau Cedex, France, September 1996.

[59] A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *ACM conference on Lisp and functional programming*, pages 196–207, Orlando, Florida, June 1994.

[60] Benjamin C. Pierce Kim B. Bruce, Luca Cardelli. Comparing object encodings. In *International Symposium on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science. Springer, September 1997.

[61] Claude Kirchner. Computing unification algorithms. In *Proceeding of the first symposium on Logic In Computer Science*, pages 206–216, Boston (USA), 1986.

[62] Claude Kirchner and François Klay. Syntactic theories and unification. CRIN & INRIA-Loraine, Nancy (France), 1989.

[63] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *Proc. 20th symp. Principles of Programming Languages*, pages 419–428. ACM press, 1993.

[64] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[65] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5) :1411–1430, September 1994.

[66] Camille le Moniès de Sagazan. Un système de types étiquetés polymorphes pour typer les calculs de processus à liaisons noms-canaux dynamiques. Thèse de doctorat, Université Paul Sabatier, Toulouse, November 1995. (Also CNRS/LAAS report 95077).

[67] Xavier Leroy. The ZINC experiment : an economical implementation of the ML language. Technical Report 117, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1990.

[68] Xavier Leroy. Type declarations, type abbreviations, and recursive types. Transaction on the newsgroup comp.lang.ml, April 30 1993.

[69] Xavier Leroy. A modular module system. Research report 2866, INRIA, April 1996.

[70] Xavier Leroy. The Objective Caml system. Software and documentation available on the Web, http ://pauillac.inria.fr/ocaml/, 1996.

[71] Xavier Leroy and Michel Mauny. The Caml Light system, version 0.5 — documentation and user's guide. Technical Report L-5, INRIA, 1992.

[72] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4) :431–463, 1993.

[73] Xavier Leroy and Pierre Weis. *Manuel de référence du langage Caml*. InterÉditions, 1993.

[74] L. Liquori and G. Castagna. A Typed Lambda Calculus of Objects. In *Proc. of ASIAN-96, International Asian Computing Science Conference*, volume 1212 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[75] Luigi Liquori. Bounded polymorphism for extensible objects. Technical Report CS-24-96, Dipartimento di Informatica, Universita' di Torino, 1997.

[76] Luigi Liquori. An Extended Theory of Primitive Objects : First Order System. In *Proceedings of ECOOP-97, International European Conference on Object Oriented Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[77] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2) :258–282, 1982.

[78] Robin Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science*, volume 230. Springer Verlag, 1980.

[79] John C. Mitchell. Coercion and type inference. In *Eleventh Annual Symposium on Principles Of Programming Languages*, 1984.

[80] John C. Mitchell. Polymorphic type inference and containment. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France)*, pages 257–278, Berlin, June 1984. Springer LNCS 173. Full version in *Information and Computation*, vol. 76, no. 2/3, 1988, pp. 211–249. Reprinted in *Logical Foundations of Functional Programming,* ed. G. Huet, Addison-Wesley (1990) 153–194.

[81] John C. Mitchell and Kathleen Fisher. A delegation-based object calculus with subtyping. In *Fundamentals of Computation Theory*, number 965 in LNCS, pages 42–61. Springer, 1995.

[82] John C. Mitchell, Furio Honsell, and Kathleen Fisher. A lambda calculus of objects and method specialization. In *IEEE Symposium on Logic in Computer Science*, pages 26–38, June 1993.

[83] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proceedings of the 23th ACM Conference on Principles of Programming Languages*, pages 54–67, January 1996.

[84] Atsushi Ohori. Extending ML polymorphism to record structure. Technical Report CSC 90/R24, University of Glasgow, Department of Computer Science, September 1990.

[85] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6) :844–895, 1996.

[86] Atsushi Ohori and Peter Buneman. Type inference in a database langage. In *ACM Conference on LISP and Functional Programming*, pages 174–183, 1988.

[87] Jens Palsberg. Efficient type inference of object types. In *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 186–195, Paris, France, July 1994. IEEE Computer Society Press. To appear in Information and Computation.

[88] Jens Palsberg and Trevor Jim. Type inference of object types with variances. Private Discussion, 1996.

[89] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988. ACM Press.

[90] Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1,2) :185–199, 1993. Preliminary version available as Technical Report CMU-CS-92-105, School of Computer Science, Carnegie Mellon University, January 1992.

[91] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, Pittsburg, Pensylvania, February 1991.

[92] Benjamin C. Pierce. Mutable objects. Unpublished note, June 1993.

[93] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2) :207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[94] Benjamin C. Pierce and David N. Turner. Local type argument synthesis with bounded quantification. Technical Report 495, Computer Science Department, Indiana University, 1997.

[95] Benjamin C. Pierce and David N. Turner. Pict : A programming language based on the pi-calculus. Technical report, Computer Science Department, Indiana University, 1997.

[96] Benjamin C. Pierce and David N. Turner. Local type inference. In *25th Symposium on Principles of Programming Languages*, pages 252–265, Sans Diego, CA, January 1998. Full version available as Indiana University CSCI Technical Report 493.

[97] François Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 122–133, January 1996.

[98] François Pottier. *Synthèse de types en présence de sous-typage : de la théorie à la pratique*. PhD thesis, Université Paris VII, July 1998.

[99] Didier Rémy. Records and variants as a natural extension of ML. In *Sixteenth Annual Symposium on Principles Of Programming Languages*, 1989.

[100] Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels.* Thèse de doctorat, Université de Paris 7, 1990.

[101] Didier Rémy. Efficient representation of extensible records. In *Proceedings of the 1992 workshop on ML and its Applications*, page 12, San Francisco, USA, June 1992.

[102] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992.

[103] Didier Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM press.

[104] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1993.

[105] Didier Rémy. Type declarations, type abbreviations, and recursive types. Transaction on the newsgroup comp.lang.ml, April 27 1993.

[106] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993.

[107] Didier Rémy. Typing record concatenation for free. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993.

[108] Didier Rémy. Programming objects with ML-ART : An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.

[109] Didier Rémy. Better subtypes and row variables for record types. Presented at the workshop on Advances in types for computer science at the Newton Institute, Cambridge, UK, August 1995.

[110] Didier Rémy. A case study of typechecking with constrained types : Typing record concatenation. Presented at the workshop on Advances in types for computer science at the Newton Institute, Cambridge, UK, August 1995.

[111] Didier Rémy. From classes to objects via subtyping. A preliminary version appeared in LNCS 1381 (ESOP 98), June 1998.

[112] Didier Rémy. From classes to objects via subtyping. In *European Symposium On Programming*, volume To appear of *Lecture Notes in Computer Science*. Springer, March 1998.

[113] Didier Rémy and Jérôme Vouillon. Objective ML : An effective object-oriented extension to ML. *Theoretical And Practice of Objects Systems*, To appear, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[114] John H. Reppy and Jon G. Riecke. Classes in Object ML. Presented at the FOOL'3 workshop, July 1996.

[115] John H. Reppy and Jon G. Riecke. Simple objects for Standard ML. In *Programming Language Design and Implementation 1996.* ACM Press, may 1996.

[116] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.

[117] Jon G. Riecke and Christopher A. Stone. Privacy via subsumption. In *Informal Proceedings of the FOOL 5 workshop on Foundations of Object Oriented Programming*, San Diego, CA, January 1998.

[118] Jérôme Vouillon. Using modules as classes. In *Informal proceedings of the FOOL'5 workshop*, 1998. Avalaible electronically at http ://pauillac.inria.fr/ remy/fool/.

[119] Mitchell Wand. Complete type inference for simple objects. In D. Gries, editor, *Second Symposium on Logic In Computer Science*, pages 207–276, Ithaca, New York, June 1987. IEEE Computer Society Press.

[120] Mitchell Wand. Corrigendum : Complete type inference for simple objects. In *Third Symposium on Logic In Computer Science*, 1988.

[121] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92–97, Pacific Grove, CA, June 1989.

[122] Pierre Weis. *The CAML Reference Manual.* INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1989.

[123] J. B. Wells. Typability and type checking in the second order λ-calculus are equivalent and undecidable. In *Ninth annual IEEE symposium on Logic in computer science*, pages 176–185, Paris, France, July 1994.

[124] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report 93–200, Rice University, February 1993.

[125] Jan Zwanenburg. A type system for record concatenation and subtyping. In *Electronic Proceedings of the FOOL 3 workshop on Foundations of Object Oriented Programming*, New Brunswick, NJ, July 1996.

Résumé

Les enregistrements, produits à champ nommés, sont une structure simple et fondamentale en programmation, et sont présents depuis longtemps dans de nombreux langages. Toutefois, certaines opérations sur les enregistrements, comme l'ajout de champs, restent délicates dans un langage fortement typé. Les objets sont, au contraire, un concept très évolué, expressif, mais les difficultés à les typer ou à les coder dans un lambda-calcul typé semblent refléter une complexité intrinsèque.

Une technique simple et générale permet d'étendre le typage des enregistrements aux opérations les plus avancées, telles que l'accès polymorphe, l'extension, la possibilité d'avoir des valeurs par défaut et une forme de concaténation. En ajoutant à ces opérations des types existentiels, objets et classes deviennent directement programmables, sans sacrifice pour leur expressivité, mais au détriment de la lisibilité des types synthétisés.

Une extension de ML avec des objets primitifs, Objective ML, à la base de la couche objet du langage Ocaml, est alors proposée. L'utilisation de constructions primitives permet, en particulier, l'abréviation automatique des types qui rend l'interface avec l'utilisateur conviviale. Une extension harmonieuse du langage avec des méthodes polymorphes est également possible.

Tout en expliquant l'imbrication entre les enregistrements, les objets et classes, ces travaux montrent surtout que le polymorphisme de ML, un concept simple et fondamental suffit à rendre compte des opérations les plus complexes sur les objets. La simplicité et la robustesse d'Objective ML et de son mécanisme de typage, qui ne sacrifient en rien l'expressivité, contribuent à démystifier la programmation avec objets, et la rendent accessible en toute sécurité à l'utilisateur, même novice.

Mots clés

Objet, Enregistrement, Typage, Polymorphisme, Synthèse de types, Variable de rangée, ML, Ocaml, Classe, Héritage, Héritage multiple, Sous-typage.

Keywords

Object, Overriding, Record, Typechecking, Polymorphism, Type inference, Row variable, ML, Ocaml, Class, Inheritance, Multiple Inheritance, Subtyping.