Chapter 7

Programmation par objets et classes

La conception d'un système de typage statique sûr pour la programmation par objets pose de sérieux problèmes. On se fixe comme objectif de garantir statiquement l'absence d'erreurs "message not understood" correspondant à évaluer $obj\#m$ où obj est un objet ne possédant pas la méthode m. (Ceci s'ajoute bien sûr aux garanties habituelles de sûreté, comme p.ex. qu'on élimine `1(2)` ou `1 + "foo"`.) Les difficultés proviennent des nombreux traits originaux des objets et des classes, dont il faut rendre compte:

- "self" et la liaison tardive;
- encapsulation des variables d'instance (accessibles seulement aux méthodes de l'objet);
- le sous-typage entre types d'objets et ses interactions avec l'inférence de types;
- l'héritage entre classes et ses liens avec le sous-typage.

Une difficulté majeure est de combiner inférence de types et subsomption implicite. La *subsomption* est l'acte de considérer une expression a de type τ comme étant d'un super-type τ' de τ. Elle est *implicite* si le changement de type (de τ à τ') n'est pas marqué dans le texte du programme, et *explicite* sinon — comme par exemple en Objective Caml où la subsomption doit être écrite sous forme d'une coercion $(a : \tau :> \tau')$. On distingue les trois combinaisons suivantes:

1. Subsomption implicite, typage explicite, pas d'inférence de types. On déclare les types des paramètres des fonctions et des variables locales. C'est l'approche suivie par la plupart des langages orientés-objets classiques: Java, Eiffel, Modula-3, C++.

2. Subsomption explicite, inférence de types à la ML. C'est l'approche suivie en Objective Caml.

3. Subsomption implicite, inférence de types par contraintes de sous-typage (généralisant l'inférence à la ML). Cette approche est le résultat de travaux récents de recherche et n'a pas encore été intégrée dans un langage complet. On ne sait pas encore si elle est réaliste (les types sont excessivement gros, p.ex.).

Dans ce cours, nous étudions (2) en détails, car c'est une application directe des rangées de types que nous avons déjà utilisées au chapitre 6. Nous verrons aussi l'approche (3) dans un cadre simplifié. Les approches (1) et (3) sont détaillées dans le cours d'option de G. Castagna et F. Pottier.

7.1 Un calcul d'objets sans classes

L'approche d'Objective Caml est de traiter les objets comme des enregistrements polymorphes dont chaque champ correspond à une méthode de l'objet. Les variables d'instance et le paramètre `self` sont traités par une sémantique d'auto-application (*self-application semantics*).

7.1.1 Syntaxe

On commence par étendre mini-ML avec des objets mais pas de classes. La construction d'un objet se fait donc en listant ses méthodes et ses variables d'instance.

Expressions:	$a ::= x \mid c \mid op \mid \texttt{fun}\ x \to a \mid a_1\ a_2$	
	$\mid (a_1, a_2) \mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2$	comme d'habitude
	$\mid \texttt{obj}(x)\langle \ldots ; \texttt{val}\ x_i = a_i; \ldots ; \texttt{method}\ m_j = a'_j; \ldots \rangle$	construction d'un objet
Opérateurs:	$op ::= \ldots \mid \#m$	sélection de la méthode m

On note $a\#m$ pour l'application d'opérateur $\#m\ a$. Ceci correspond à l'appel de la méthode m de l'objet a.

L'identificateur x dans la construction $\texttt{obj}(x)\langle\ldots\rangle$ correspond au paramètre “self” des méthodes. Une méthode de cet objet peut donc faire $x\#m$ pour rappeler une autre méthode du même objet.

Les variables d'instance sont considérées comme immuables. On peut y mettre des références comme au chapitre 4 pour modéliser les variables d'instance mutables.

7.1.2 Règles de réduction

La règle de réduction pour $\#m$ est la suivante:

$$\begin{array}{lll} v\#m_j & \xrightarrow{\varepsilon} & a_j[s \leftarrow v,\ x_1 \leftarrow v_1,\ \ \ldots, x_n \leftarrow v_n] \\ & & \text{si } v = \texttt{obj}(s)\langle \texttt{val}\ x_i = v_i; \ldots ; \texttt{val}\ x_n = v_n; \texttt{method}\ m_1 = a_1; \ldots ; \texttt{method}\ m_k = a_k \rangle \end{array}$$

L'auto-application est visible ici par le remplacement de l'identificateur s par l'objet v lui-même dans le corps a_j de la méthode m_j. De la sorte, les appels $s\#m'$ présents dans a_j feront bien référence à l'objet v lui-même. Remarquons que l'auto-application est une forme de récursion. Par exemple, il est facile de définir des fonctions récursives ou mutuellement récursives à l'aide d'un objet:

```
let o = obj(s) < method fact = fun n →
                                  if n = 0 then 1 else n * s#fact (n-1) >
in o#fact 10
```

Les valeurs et les contextes d'évaluation sont:

Valeurs:

$$\begin{array}{l} v ::= c \mid op \mid \texttt{fun}\ x \to a \mid (v_1, v_2) \\ \quad\ \mid \texttt{obj}(s)\langle \texttt{val}\ x_1 = v_1; \ldots ; \texttt{val}\ x_n = v_n; \texttt{method}\ m_1 = a_1; \ldots ; \texttt{method}\ m_k = a_k \rangle \end{array}$$

Contextes:

$$\Gamma ::= [\,] \mid \Gamma\ a \mid v\ \Gamma \mid \texttt{let}\ x = \Gamma\ \texttt{in}\ a \mid (\Gamma, a) \mid (v, \Gamma)$$
$$\mid \texttt{obj}(s)\langle \ldots \texttt{val}\ x_{i-1} = v_i; \texttt{val}\ x_i = \Gamma; \texttt{val}\ x_{i+1} = a_{i+1}; \ldots; \texttt{method}\ m_j = a'_j; \ldots; \rangle$$

Notons que dans un objet complètement évalué, les variables d'instance sont complètement évaluées, mais pas les corps des méthodes. En ce sens, la partie "variables d'instance" d'un objet se comporte comme un n-uplet ou un enregistrement, alors que la partie "méthode" se comporte comme un corps de fonction (de paramètre s).

7.1.3 L'algèbre de types

Le type d'un objet est essentiellement identique à celui d'un enregistrement: il liste les noms des méthodes de l'objet, avec pour chaque méthode le type de son résultat. Nous conservons les mêmes mécanismes de rangées et d'annotations de présence que pour les types d'enregistrements.

Types:	$\tau ::= \alpha \mid T \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$	comme précédemment
	$\mid \langle \tau \rangle$	type d'objet
	$\mid \emptyset$	la rangée vide
	$\mid m : \tau_1;\ \tau_2$	la rangée contenant $m : \tau_1$ plus ce que contient la rangée τ_2
	$\mid \texttt{Abs}$	le champ est absent (indéfini)
	$\mid \texttt{Pre}\ \tau$	le champ est présent (défini) avec le type τ

Exemples: le type $\langle m : \texttt{Pre int}; n : \texttt{Pre string}; \emptyset \rangle$ est le type des objets possédant une méthode m renvoyant un entier, une méthode n renvoyant une chaîne, et aucune autre méthode. De tels types "fermés" apparaissent lorsque l'on type la création d'un objet $\texttt{obj}(s)\langle \ldots \rangle$.

Le type $\langle m : \texttt{Pre int};\ \alpha \rangle$ est le type des objets possédant une méthode m renvoyant un entier, et éventuellement d'autres méthodes (dont le type va instancier α). De tels types "ouverts" apparaissent naturellement pour les paramètres de fonctions; ainsi,

```
fun obj → 1 + obj#m
```

a le type $\langle m : \texttt{Pre int};\ \alpha \rangle \to \texttt{int}$.

Théorie équationnelle: comme pour les enregistrements, on considère les types modulo les équations de commutativité et d'absorption:

$$\begin{array}{rcll} m_1 : \tau_1;\ m_2 : \tau_2;\ \tau & = & m_2 : \tau_2;\ m_1 : \tau_1;\ \tau & \text{(commutativité)} \\ \emptyset & = & m : \texttt{Abs};\ \emptyset & \text{(absorption)} \end{array}$$

Sortes: comme pour les enregistrements, on utilise des sortes pour éviter les types absurdes $\langle m : \texttt{int}; \texttt{bool} \rangle$ ou contradictoires $\langle m : \texttt{Pre int};\ m : \texttt{Pre bool};\ \emptyset \rangle$. Le système de sortage est identique à celui des enregistrements (section 6.3.3).

7.1.4 Règles de typage

$$\frac{\tau \leq E(x)}{E \vdash x : \tau} \text{ (var-inst)} \qquad \frac{\tau \leq TC(c)}{E \vdash c : \tau} \text{ (const-inst)} \qquad \frac{\tau \leq TC(op)}{E \vdash op : \tau} \text{ (op-inst)}$$

$$\frac{\vdash \tau_1 :: \texttt{TYPE} \qquad E + \{x : \tau_1\} \vdash a : \tau_2}{E \vdash (\texttt{fun}\ x \to a) : \tau_1 \to \tau_2} \text{ (fun)} \qquad \frac{E \vdash a_1 : \tau' \to \tau \qquad E \vdash a_2 : \tau'}{E \vdash a_1\ a_2 : \tau} \text{ (app)}$$

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2} \text{ (paire)} \qquad \frac{E \vdash a_1 : \tau_1 \qquad E + \{x : Gen(\tau_1, E)\} \vdash a_2 : \tau_2}{E \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \tau_2} \text{ (let-gen)}$$

$$\frac{\begin{array}{c} i \neq j \Rightarrow m_i \neq m_j \qquad E \vdash a_i : \tau_i \\ E + \{x : \langle m_1 : \texttt{Pre}\ \tau'_1;\ \ldots;\ m_k : \texttt{Pre}\ \tau'_k;\ \emptyset\rangle;\ x_i : \tau_i\} \vdash a'_j : \tau'_j \end{array}}{E \vdash \texttt{obj}(x)\langle \ldots; \texttt{val}\ x_i = a_i; \ldots; \texttt{method}\ m_j = a'_j; \ldots\rangle : \langle m_1 : \texttt{Pre}\ \tau'_1;\ \ldots;\ m_k : \texttt{Pre}\ \tau'_k;\ \emptyset\rangle} \text{ (objet)}$$

Enfin, le schéma de types pour l'opérateur $\#m$ est bien sûr:

$$\#m \quad : \quad \forall \alpha, \beta.\ \langle m : \texttt{Pre}\ \alpha;\ \beta\rangle \to \alpha$$

7.1.5 Sûreté du typage

La sûreté du typage ci-dessus se montre comme au chapitre 2. En particulier, il faut montrer les hypothèses (H1) et (H2) pour l'opérateur $\#m$. Pour (H1), supposons $v\#m \xrightarrow{\varepsilon} a'$ et $E \vdash v\#m : \tau$, avec $v = \texttt{obj}(s)\langle \ldots; \texttt{val}\ x_i = v_i; \ldots; \texttt{method}\ m_j = a_j; \ldots\rangle$, $m = m_j$, et $a' = a_j[s \leftarrow v,\ x_i \leftarrow v_i]$. On a forcément une dérivation de typage de la forme:

$$\frac{\dfrac{i \neq j \Rightarrow m_i \neq m_j \qquad E \vdash v_i : \tau'_i \qquad E + \{s : \tau_s;\ x_i : \tau'_i\} \vdash a_k : \tau_k}{E \vdash v : \tau_s}}{E \vdash v\#m_j : \tau_j}$$

avec $\tau = \tau_j$ et $\tau_s = \langle \ldots; m_i : \texttt{Pre}\ \tau_i; \ldots; \emptyset\rangle$. On a donc $E \vdash v_i : \tau'_i$ pour tout i, et de plus $E \vdash v : \tau_s$. Appliquant un lemme de substitution semblable au lemme 2.2 à $E + \{s : \tau_s;\ x_i : \tau'_i\} \vdash a_j : \tau_j$, il vient $E \vdash a_j[s \leftarrow v,\ x_i \leftarrow v_i] : \tau_j$, c'est-à-dire $E \vdash a' : \tau_j$.

La preuve de (H2) est du même style; nous l'omettons.

7.1.6 Inférence de types

L'inférence de types pour le système de types de cette section est très proche de celle pour mini-ML avec enregistrements (section 6.5). On utilise un algorithme d'unification dans le style de la section 6.5.1, et un algorithme W modifié avec le cas suivant pour la construction d'objets:

- Si a est $\texttt{obj}(x)\langle \texttt{val}\ x_i = a_i; \texttt{method}\ m_j = a'_j\rangle$:
  vérifier que les noms de méthodes m_i sont deux à deux distincts
  soit $(\vec{\tau}, \varphi_1, V_1) = \vec{W}(E, \vec{a}, V)$
  soit $\alpha \in V_1$ une variable de sorte `TYPE`

soit $(\vec{\tau'}, \varphi_2, V_2) = W(\varphi_1(E) + \{\texttt{self}\ x : \alpha; \texttt{val}\ x_i : \tau_i\}, \vec{a'}, V_1 \setminus \{\alpha\})$
soit $\mu = \texttt{mgu}\{\varphi_2(\alpha) \stackrel{?}{=} \langle m_1 : \texttt{Pre}\ \tau'_1;\ \ldots;\ m_k : \texttt{Pre}\ \tau'_k;\ \emptyset\rangle\}$
prendre $\tau = \mu(\varphi_2(\alpha))$ et $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ et $V = V_2 \setminus \mathrm{Dom}(\mu)$.

On a noté $\vec{W}$ l'inférence de types pour une séquence d'expressions: $(\vec{\tau}, \varphi, V') = \vec{W}(E, \vec{a}, V)$ est défini par:

soit $(\tau'_1, \varphi_1, V_1) = W(E, a_1, V)$
soit $(\tau'_2, \varphi_2, V_2) = W(\varphi_1(E), a_2, V_1)$
soit ...
soit $(\tau'_n, \varphi_n, V_n) = W((\varphi_{n-1} \circ \cdots \circ \varphi_1)(E), a_n, V_{n-1})$
prendre $\tau_i = (\varphi_n \circ \cdots \circ \varphi_{i+1})(\tau'_i)$ pour $i = 1 \ldots n$, et $\varphi = \varphi_n \circ \cdots \circ \varphi_1$ et $V' = V_n$.

7.2 Sous-typage et subsomption explicite

Le calcul d'objets ci-dessus n'a aucun mécanisme de sous-typage. En particulier, il n'est pas possible d'oublier des méthodes dans un objet: si $a : \langle \texttt{m} : \texttt{Pre int};\ \emptyset\rangle$ et $b : \langle \texttt{m} : \texttt{Pre int};\ \texttt{n} : \texttt{Pre string};\ \emptyset\rangle$, l'expression **if** *cond* **then** a **else** b n'est pas typable, et en particulier n'a pas le type "naturel" $\langle \texttt{m} : \texttt{Pre int};\ \emptyset\rangle$.

Pour pallier cette faiblesse, on ajoute une famille d'opérateurs de coercion $\texttt{coerce}_{\tau,\tau'}$ qui permettent de transformer explicitement un objet de type τ en un objet du super-type τ'.

Opérateurs: $\quad op ::= \texttt{coerce}_{\tau,\tau'} \quad$ pour tous τ, τ' tels que $\tau <: \tau'$

La construction $(a : \tau >: \tau')$ d'Objective Caml est bien sûr du sucre syntaxique pour $\texttt{coerce}_{\tau,\tau'}(a)$. Le type des opérateurs de coercion et leur règle de réduction sont:

$$\begin{array}{rcl} \texttt{coerce}_{\tau,\tau'} & : & \forall \vec{\alpha}.\ \tau \to \tau' \quad \text{si } \tau <: \tau' \text{ et } \vec{\alpha} = \mathcal{L}(\tau) \cup \mathcal{L}(\tau') \\ \texttt{coerce}_{\tau,\tau'}(v) & \stackrel{\varepsilon}{\to} & v \end{array}$$

Remarquez que ces opérateurs `coerce` généralisent très naturellement les contraintes de types de ML comme présentées section 4.4.

Relation de sous-typage: La relation de sous-typage $<:$ qui caractérise les opérateurs `coerce` valides est définie par les règles d'inférence suivantes:

$$T <: T \qquad \alpha <: \alpha \qquad \frac{\tau' <: \tau \quad \varphi <: \varphi'}{(\tau \to \varphi) <: (\tau' \to \varphi')} \qquad \frac{\tau <: \tau' \quad \varphi <: \varphi'}{(\tau \times \varphi) <: (\tau' \times \varphi')}$$

$$\frac{\tau <: \tau'}{\langle\tau\rangle <: \langle\tau'\rangle} \qquad \tau <: \emptyset \qquad \frac{\tau <: \tau' \quad \varphi <: \varphi'}{(m : \tau;\ \varphi) <: (m : \tau';\ \varphi')}$$

$$\texttt{Abs} <: \texttt{Abs} \qquad \frac{\tau <: \tau'}{\texttt{Pre}\ \tau <: \texttt{Pre}\ \tau'}$$

Deux types de base ou deux variables de types sont en relation de sous-typage si et seulement si ils sont égaux.

Un type objet $\langle\tau\rangle$ est sous-type d'un autre $\langle\tau'\rangle$ si toutes les méthodes mentionnées dans la rangée τ' sont également mentionnées dans τ, et leur type dans τ est sous-type de celui dans τ'. Si τ' se termine par $\emptyset$, la rangée τ peut aussi mentionner des méthodes supplémentaires qui n'apparaissent pas dans τ' (en raison de l'axiome $\tau <: \emptyset$). En revanche, si τ' se termine par une variable de rangée, τ doit se terminer par la même variable et mentionner les mêmes méthodes que τ'.

Pour les types produit, il y a sous-typage si les types des premières composantes sont sous-types, ainsi que les types des secondes composantes. On dit que le produit est un constructeur de type *covariant* en ses deux arguments. Autrement dit, les deux fonctions des types dans les types $\_ \times \tau$ et $\tau \times \_$ sont des fonctions croissantes pour l'ordre $<:$.

Enfin, pour le sous-typage entre types flèches, on a covariance en le type du résultat, mais *contravariance* en le type de l'argument: les types arguments doivent être en sous-typage dans l'ordre inverse du sous-typage entre les types flèche. Autrement dit, la fonction $\_ \rightarrow \tau$ est décroissante, alors que la fonction $\tau \rightarrow \_$ est croissante. Ceci se comprend mieux en pensant aux types comme à des ensembles de valeurs, et à la relation de sous-typage comme à l'inclusion entre ensembles de valeurs. En théorie des ensembles, on a aussi que $A \rightarrow B$ (l'ensemble des fonctions de A dans B) est inclus dans $A' \rightarrow B'$ (l'ensemble des fonctions de A' dans B') si et seulement si $A' \subseteq A$ et $B \subseteq B'$.

Exemples: on a les sous-typages et les non-sous-typages suivants:

$$\begin{array}{rcl}
\langle m : \texttt{Pre int};\ \emptyset\rangle & <: & \langle\emptyset\rangle \\
\langle o : \langle m : \texttt{Pre int};\ \emptyset\rangle\rangle & <: & \langle o : \langle\emptyset\rangle\rangle \\
\langle m : \texttt{Pre int};\ \alpha\rangle & \not<: & \langle\alpha\rangle \\
\texttt{int} \rightarrow \langle m : \texttt{Pre int};\ \emptyset\rangle & <: & \texttt{int} \rightarrow \langle\emptyset\rangle \\
\langle m : \texttt{Pre int};\ \emptyset\rangle \rightarrow \texttt{int} & \not<: & \langle\emptyset\rangle \rightarrow \texttt{int} \\
\langle\emptyset\rangle \rightarrow \texttt{int} & <: & \langle m : \texttt{Pre int};\ \emptyset\rangle \rightarrow \texttt{int}
\end{array}$$

Sous-typage et sûreté du typage: la règle de réduction pour $\texttt{coerce}_{\tau,\tau'}$ compromet la préservation du typage pendant la réduction: en effet, si v a le type τ, l'expression $\texttt{coerce}_{\tau,\tau'}(v)$ a le type τ', mais se réduit en v qui a le type $\tau \neq \tau'$.

Une manière de retrouver la préservation du typage et donc de montrer la sûreté du typage est d'ajouter (pour les besoins de la preuve de sûreté uniquement) une règle de subsomption implicite:

$$\frac{E \vdash a : \tau \qquad \tau <: \tau'}{E \vdash a : \tau'}\ \text{(sub)}$$

Avec cette règle, on a bien que la réduction $\texttt{coerce}_{\tau,\tau'}(v) \xrightarrow{\varepsilon} v$ préserve le typage, puisque v qui a le type τ par hypothèse a aussi le type τ' par application de la règle (sub).

Bien sûr, cette règle (sub) rend l'inférence de types problématique — c'est pour éviter cela que nous mettons des coercions explicites! Il faut donc typer les programmes source sans la règle (sub), ce qui permet de faire de l'inférence de types, et n'ajouter la règle (sub) que pour raisonner sur la préservation du typage pendant la réduction.

7.3 Classes

Nous ajoutons maintenant au calcul d'objets une notion de classes avec un mécanisme d'héritage. Les classes sont des collections de définitions de méthodes et de variables d'instance. Une construction `new` explicite permet de prendre une instance d'une classe, c'est-à-dire de construire un objet ayant les méthodes et les variables indiquées dans la classe. Pour simplifier, nous présentons les classes comme faisant partie des expressions; dans un langage réaliste, on a tendance à distinguer un sous-langage de classes distinct du langage des expressions (dans un langage comme Objective Caml, cela simplifie l'inférence de types).

Autres simplifications: nous traiterons seulement les classes ne contenant pas de variables d'instances, uniquement des méthodes. Nous ne traitons pas non plus l'héritage multiple. (Voir l'article de Rémy et Vouillon référencé à la fin de ce chapitre pour un traitement plus complet.)

Expressions:	$a ::= \ldots$	
	$\mid \texttt{new}(a)$	création d'un objet
	$\mid \texttt{class}(x)\langle \ldots m_i = a_i \ldots \rangle$	définition de classe
	$\mid \texttt{class}(x)\langle \texttt{inherit}(a);\ m = a \rangle$	héritage et ajout ou redéfinition d'une méthode
Types:	$\tau ::= \ldots \mid \texttt{class}(\tau_1)\ \tau_2$	type de classe

7.3.1 Évaluation des classes

L'évaluation des classes consiste à résoudre l'héritage. Pour ce faire, on évalue la classe héritée, et on remplace la méthode redéfinie si nécessaire.

$$\texttt{new}(\texttt{class}(x)\langle m_i = a_i \rangle) \xrightarrow{\varepsilon} \texttt{obj}(x)\langle \texttt{method}\ m_i = a_i \rangle$$

$$\texttt{class}(x)\langle \texttt{inherit}(\texttt{class}(y)\langle m_i = a_i \rangle_{i=1\ldots n});\ m = a \rangle \xrightarrow{\varepsilon} \texttt{class}(y)\langle (m_i = a_i)_{i=1\ldots n, m_i \neq m};\ m = a[x \leftarrow y] \rangle$$

Valeurs: $v ::= \ldots \mid \texttt{class}(x)\langle m_i = a_i \rangle$

Contextes: $\Gamma ::= \ldots \mid \texttt{new}(\Gamma) \mid \texttt{class}(x)\langle \texttt{inherit}\ \Gamma;\ m = a \rangle$

7.3.2 Typage des classes

Le type d'une classe est $\texttt{class}(\tau_1)\ \tau_2$, où τ_1 est le type du paramètre "self" des méthodes, et τ_2 est un type d'objet représentant les types des méthodes définies dans la classe. Le cas $\tau_1 = \tau_2$ correspond à une classe "autosuffisante", c'est-à-dire qui définit toutes les méthodes qu'elle utilise dans des appels via "self". On peut donc faire `new` sur une telle classe pour construire un objet avec ses méthodes. Le cas où τ_1 mentionne des méthodes qui n'apparaissent pas dans τ_2 correspond à une classe virtuelle en Objective Caml: certaines méthodes restent à définir. Par exemple, la classe OCaml

```
class virtual c =
  object(self)
```

```
    method virtual m : int
    method n = 1 + self#m
  end
```

reçoit ici le type $\texttt{class}(\langle \texttt{m} : \texttt{Pre int};\ \alpha\rangle)\ \langle \texttt{n} : \texttt{Pre int}; \emptyset\rangle$.

$$\frac{E \vdash a : \texttt{class}(\tau)\ \tau}{E \vdash \texttt{new}(a) : \tau}\ (\text{new})$$

$$\frac{E + \{x : \tau\} \vdash a_i : \tau_i}{E \vdash \texttt{class}(x)\langle m_i = a_i\rangle : \texttt{class}(\tau)\ \langle m_1 : \texttt{Pre}\ \tau_1;\ \ldots;\ m_k : \texttt{Pre}\ \tau_k;\ \emptyset\rangle}\ (\text{classe})$$

$$\frac{E \vdash a_1 : \texttt{class}(\tau_x)\ \langle m : \texttt{Abs};\ \tau\rangle \qquad E + \{x : \tau_x\} \vdash a_2 : \tau'}{E \vdash \texttt{class}(x)\langle \texttt{inherit}(a_1);\ m = a_2\rangle : \texttt{class}(\tau_x)\ \langle m : \texttt{Pre}\ \tau';\ \tau\rangle}\ (\text{exten})$$

$$\frac{E \vdash a_1 : \texttt{class}(\tau_x)\ \langle m : \texttt{Pre}\ \tau';\ \tau\rangle \qquad E + \{x : \tau_x\} \vdash a_2 : \tau'}{E \vdash \texttt{class}(x)\langle \texttt{inherit}(a_1);\ m = a_2\rangle : \texttt{class}(\tau_x)\ \langle m : \texttt{Pre}\ \tau';\ \tau\rangle}\ (\text{redéf})$$

Pour l'héritage, on a deux règles suivant que l'on ajoute une nouvelle méthode ou que l'on redéfinit une méthode existant dans la classe héritée. Dans le second cas, la nouvelle définition de la méthode doit avoir le même type que dans la super-classe.

Dans tous les cas d'héritage, le type de "self" doit être le même dans la nouvelle classe et dans la classe héritée. Cela garantit que les méthodes héritées qui retournent "self" en résultat restent bien typées. Ceci ne restreint pas l'expressivité du langage, car en général la classe dont on hérite est liée à un identificateur, et a donc un schéma de type de la forme

$$\forall\alpha.\ \texttt{class}(\langle m_i : \tau_i;\ \alpha\rangle)\ \langle m_j : \tau_j;\ \emptyset\rangle$$

En instanciant correctement la variable de rangée α, on peut facilement satisfaire les contraintes des règles (exten) et (redéf).

Le même effet de polymorphisme sur les types de classes s'applique aussi à la spécialisation de "selftype" entre une classe et la classe dont elle hérite.

Exemple: voici un fragment d'Objective Caml qui illustre plusieurs points délicats: classe paramétrée par un type; spécialisation de ce paramètre dans une sous-classe; utilisation de "selftype" pour typer une méthode qui renvoie "self".

```
class ['a] comparable =
  object (self : 'selftype)
    method virtual leq : 'selftype -> bool
    method min y = if self#leq y then self else y
  end
class int_comparable n =
  object(self)
    inherit [int] comparable
```

```
    method get = n
    method leq obj = self#get <= obj#get
  end
```

La traduction dans notre petit langage de classes donne:

```
let comparable =
  class(self) < min = fun y → if self#leq y then self else y > in
let int_comparable =
  fun n →
    class(self) < inherit(comparable);
                  method get = n;
                  method leq = fun obj → self#get <= obj#get >
```

Cet exemple est typable avec les schémas de types suivants:

$$\begin{array}{rcl}
\texttt{comparable} & : & \forall\beta.\ \texttt{class}(\tau)\ \langle\texttt{min} : \texttt{Pre}\ (\tau \to \tau);\ \emptyset\rangle \\
& & \text{avec } \tau = \mu\alpha.\ \langle\texttt{leq} : \texttt{Pre}(\alpha \to \texttt{bool});\ \beta\rangle \\
\texttt{int_comparable} & : & \forall\beta.\ \texttt{class}(\tau')\ \langle\texttt{min} : \texttt{Pre}(\tau' \to \tau');\ \texttt{get} : \texttt{Pre int};\ \texttt{leq} : \texttt{Pre}\ (\tau' \to \texttt{bool});\ \emptyset\rangle \\
& & \text{avec } \tau' = \mu\alpha.\ \langle\texttt{leq} : \texttt{Pre}(\alpha \to \texttt{bool});\ \texttt{get} : \texttt{Pre int};\ \beta\rangle
\end{array}$$

Une instance du schéma de `int_comparable` est $\texttt{class}(\tau'')\ \tau''$, avec $\tau'' = \langle\texttt{min} : \texttt{Pre}(\tau'' \to \tau'');\ \texttt{get} : \texttt{Pre int};\ \texttt{leq} : \texttt{Pre}(\tau'' \to \texttt{bool});\ \emptyset\rangle$, ce qui montre que l'on peut faire `new(int_comparable)`.

7.4 Les types récursifs

La programmation par objets fait rapidement apparaître le besoin de types *récursifs* (infinis, cycliques). Par exemple, considérons une méthode `min` qui renvoie le plus petit de l'objet lui-même et d'un autre objet passé en paramètre:

```
obj(s) < val clé = une expression entière;
         method rang = clé;
         method min o = if s#rang <= o#rang then s else o >
```

Cet objet n'est pas typable dans le système de la section 7.1. En effet, le type de l'objet doit avoir la forme suivante:

$$\tau_s = \langle\texttt{rang} : \texttt{Pre int};\ \texttt{min} : \texttt{Pre}\ (\tau_o \to \tau_o);\ \ldots\rangle$$

mais pour que le `if...then...else` soit bien typé, il faut $\tau_s = \tau_o$, et c'est impossible avec notre algèbre de types car τ_o est un sous-terme strict de τ_s.

Pour contourner cette restriction, on peut introduire des types récursifs dans l'algèbre de types. Intuitivement, un type récursif est une expression de type infinie, comme par exemple $\texttt{int} \to \texttt{int} \to \texttt{int} \to \ldots$. Avec les types récursifs, une équation comme $\alpha \stackrel{?}{=} \tau[\alpha]$ admet une solution qui est le type infini $\tau[\tau[\tau[\ldots]]]$.

7.4.1 Présentations des types récursifs

Il y a plusieurs manières de formaliser rigoureusement les types récursifs:

Limites de suites de types finis: on voit les types récursifs comme des limites de suites d'approximations finies. Ainsi, $\texttt{int} \to \texttt{int} \to \texttt{int} \to \dots$ est la limite de la suite $\texttt{int}$, $(\texttt{int} \to \texttt{int})$, $(\texttt{int} \to \texttt{int} \to \texttt{int})$, ... L'exercice 7.5 explore cette approche.

Arbres infinis rationnels: de même que les expressions de types normales peuvent être vues comme des arbres finis (avec des nœuds étiquetés par T, $\to$, $\times$, α, ...), on peut voir les types récursifs comme des arbres infinis. On impose que ces arbres soient rationnels, c'est-à-dire comportent un nombre fini de sous-arbres différents.

Graphes: on représente les types comme des graphes finis dont les nœuds sont étiquetés par T, $\to$, $\times$, α, ... Les types normaux correspondent à des graphes acycliques; les types récursifs, à des graphes comportant des cycles. L'exercice 7.4 explore cette approche.

Présentation syntaxique avec des μ-types: on enrichit les expressions de types comme suit:

Types: $\tau ::= \dots \mid \mu\alpha.\ \tau \quad \text{si } \tau \neq \alpha$

Le type $\mu\alpha.\ \tau$ est une représentation finie du seul type τ' (fini ou infini) qui vérifie l'égalité $\tau' = \tau[\alpha \leftarrow \tau']$.

Par exemple, $\mu\alpha.\texttt{int} \to \alpha$ est le type infini $\texttt{int} \to \texttt{int} \to \texttt{int} \to \dots$. C'est la seule solution de l'équation $\tau' = \texttt{int} \to \tau'$.

Dans l'exemple `min` ci-dessus, un type correct pour l'objet est

$$\mu\alpha.\langle \texttt{rang} : \texttt{Pre int};\ \texttt{min} : \texttt{Pre}(\alpha \to \alpha);\ \emptyset\rangle$$

Les types μ sont identifiés modulo la théorie équationnelle suivante:

$$\mu\alpha.\tau = \mu\beta.\tau[\alpha \leftarrow \beta] \ \text{(renommage)} \qquad\qquad \mu\alpha.\tau = \tau[\alpha \leftarrow \mu\alpha.\tau] \ \text{(déroulage)}$$

$$\frac{\tau[\alpha \leftarrow \tau_1] = \tau[\alpha \leftarrow \tau_2] \qquad \tau \neq \alpha}{\tau_1 = \tau_2} \ \text{(unicité)}$$

L'axiome (renommage) exprime que α dans $\mu\alpha.\ \tau$ est une variable liée et peut être renommée à volonter. L'axiome (déroulage) permet d'"enrouler" et de "dérouler" un type μ à volonté pour satisfaire des égalités de types. Elle capture l'idée que $\mu\alpha.\ \tau$ est une solution de l'équation $\alpha \stackrel{?}{=} \tau$. Enfin, la règle d'inférence (unicité) exprime que $\mu\alpha.\ \tau$ est la seule solution de l'équation $\alpha \stackrel{?}{=} \tau$. Donc, si τ_1 et τ_2 satisfont cette équation, ils sont forcément égaux.

Exemple: les deux types $\mu\alpha.\ \texttt{int} \to \alpha$ et $\mu\alpha.\ \texttt{int} \to \texttt{int} \to \alpha$ sont égaux, car ils sont tous deux solution de $\alpha \stackrel{?}{=} \texttt{int} \to \texttt{int} \to \alpha$: en déroulant le premier type deux fois, on a bien

$$\mu\alpha.\ \texttt{int} \to \alpha = \texttt{int} \to (\mu\alpha.\ \texttt{int} \to \alpha) = \texttt{int} \to \texttt{int} \to (\mu\alpha.\ \texttt{int} \to \alpha)$$

et en déroulant le second une fois,

$$\mu\alpha.\ \texttt{int} \to \texttt{int} \to \alpha = \texttt{int} \to \texttt{int} \to (\mu\alpha.\ \texttt{int} \to \texttt{int} \to \alpha)$$

Présentation syntaxique sous forme de types avec équations: on manipule syntaxiquement à la fois des expressions de types et des équations entre variables de types et types. Par exemple, $\mathtt{int} \to \mathtt{int} \to \ldots$ est vu comme le type α dans le contexte de l'équation $\alpha = \mathtt{int} \to \alpha$.

Quelle que soit la présentation des types récursifs retenue, on conserve exactement les mêmes règles de typage que dans le cas non récursif. Simplement, certaines contraintes d'égalité imposées par les règles sont maintenant satisfiables par des types récursifs, alors qu'elles n'avaient pas de solution avec les types normaux.

7.4.2 Sous-typage et types récursifs

Le sous-typage en présence de types récursifs est délicat. Pour déterminer $\tau <: \tau'$, si p.ex. τ est un type μ mais pas τ', il faut bien sûr dérouler le type μ pour faire apparaître le constructeur de type qui est en dessous et le comparer avec le constructeur de tête de τ'. Mais si les deux types τ, τ' sont des types μ, dérouler systématiquement les deux types mène à des dérivations de sous-typage infinies, et donc incorrectes. Par exemple, pour déterminer si

$$\mu\alpha.\langle m : \mathtt{Pre}\ \alpha;\ \beta\rangle <: \mu\alpha.\langle m : \mathtt{Pre}\ \alpha;\ \emptyset\rangle,$$

on peut en déroulant les deux types se ramener à

$$\langle m : (\mu\alpha.\langle m : \mathtt{Pre}\ \alpha;\ \beta\rangle);\ \beta\rangle <: \langle m : (\mu\alpha.\langle m : \mathtt{Pre}\ \alpha;\ \emptyset\rangle);\ \emptyset\rangle,$$

mais pour prouver ce sous-typage, il faut avoir une dérivation de

$$\mu\alpha.\langle m : \mathtt{Pre}\ \alpha;\ \beta\rangle <: \mu\alpha.\langle m : \mathtt{Pre}\ \alpha;\ \emptyset\rangle$$

et nous voilà ramenés au problème de départ.

La solution est d'ajouter une règle de sous-typage supplémentaire pour le cas de deux types μ, disant que $\mu\alpha.\ \tau <: \mu\beta.\ \tau'$ si, en faisant l'hypothèse que $\alpha <: \beta$, on peut montrer que $\tau <: \tau'$. C'est une forme de preuve par *co-induction*: si, en supposant que les deux types μ après expansion sont sous-types, on peut montrer qu'ils sont sous-types, alors ils sont sous-types. La relation de sous-typage devient $H \vdash \tau <: \tau'$, où l'environnement H est un ensemble d'hypothèses de la forme $\alpha <: \beta$. Les règles définissant le sous-typage sont:

$$H \vdash T <: T \qquad H \vdash \alpha <: \alpha \qquad \frac{H \vdash \tau' <: \tau \quad H \vdash \varphi <: \varphi'}{H \vdash \tau \to \varphi <: \tau' \to \varphi'} \qquad \frac{H \vdash \tau <: \tau' \quad H \vdash \varphi <: \varphi'}{H \vdash \tau \times \varphi <: \tau' \times \varphi'}$$

$$\frac{H \vdash \tau <: \tau'}{H \vdash \langle\tau\rangle <: \langle\tau'\rangle} \qquad H \vdash \tau <: \emptyset \qquad \frac{H \vdash \tau <: \tau' \quad H \vdash \varphi <: \varphi'}{H \vdash (m : \tau;\ \varphi) <: (m : \tau';\ \varphi')}$$

$$\mathtt{Abs} <: \mathtt{Abs} \qquad \frac{\tau <: \tau'}{Pre\ \tau <: \mathtt{Pre}\ \tau'}$$

$$\frac{(\alpha <: \beta) \in H}{H \vdash \alpha <: \beta} \qquad \frac{H \cup \{(\alpha <: \beta)\} \vdash \tau <: \tau'}{H \vdash \mu\alpha.\ \tau <: \mu\beta.\ \tau'}$$

7.4.3 Inférence en présence de types récursifs

L'introduction de types récursifs ne modifie pas l'algorithme W, mais nécessite de remplacer l'unification entre termes finis par de l'unification entre termes rationnels, c'est-à-dire de l'unification entre graphes.

L'idée de base est de faire comme dans l'algorithme d'unification entre termes, sauf qu'on supprime le test d'occurrence dans le cas $\alpha \stackrel{?}{=} \tau$: au lieu de prendre $[\alpha \leftarrow \tau]$ si $\alpha \notin \mathcal{L}(\tau)$ et d'échouer sinon, on prend dans tous les cas $[\alpha \leftarrow \mu\alpha.\ \tau]$. Par l'équation de déroulement, on a bien $\mu\alpha.\ \tau = \tau[\alpha \leftarrow \mu\alpha.\ \tau]$. Remarquez que si $\alpha \notin \mathcal{L}(\tau)$, on retrouve la même solution que dans le cas des termes finis, car alors $\mu\alpha.\ \tau = \tau$.

La difficulté est d'assurer la terminaison de l'algorithme d'unification. Pour ce faire, il faut introduire des formalismes plus complexes que ceux que nous avons employé jusqu'ici: unification entre graphes, ou bien multi-équations. L'exercice 7.4 est une introduction à l'unification entre graphes.

7.5 Inférence par contraintes de sous-typage

Dans cette partie, nous introduisons l'approche 3 (subsomption implicite et inférence de types par contraintes de sous-typage) dans un cadre simplifié où nous n'avons pas de `let` polymorphe et pas de variables de rangées dans les types d'objets.

7.5.1 Règles de typage

Algèbre de types: on considère les types suivants:

Types:	$\tau ::= \alpha \mid T \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$	comme précédemment
	$\mid \langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle$	type d'objet
	$\mid \mu\alpha.\ \tau$	type récursif
	$\mid \top$	le super-type de tous les types
	$\mid \bot$	le type vide (sous-type de tous les types)

On suppose que dans un type d'objet, les noms de méthodes m_i sont tous différents. L'ordre des composantes $m_i : \tau_i$ dans un type d'objet n'est pas significatif.

Relation de sous-typage: La relation de sous-typage est une simplification de celle de la section 7.2, avec les types récursifs traités comme expliqué dans la section 7.4.2:

$$H \vdash \bot <: \tau \qquad H \vdash \tau <: \top \qquad H \vdash T <: T \qquad H \vdash \alpha <: \alpha$$

$$\frac{H \vdash \tau' <: \tau \quad H \vdash \varphi <: \varphi'}{H \vdash (\tau \to \varphi) <: (\tau' \to \varphi')} \qquad \frac{H \vdash \tau <: \tau' \quad H \vdash \varphi <: \varphi'}{H \vdash (\tau \times \varphi) <: (\tau' \times \varphi')}$$

$$\frac{H \vdash \tau_1 <: \varphi_1 \quad \ldots \quad H \vdash \tau_k <: \varphi_k}{H \vdash \langle m_1 : \tau_1; \ldots; m_k : \tau_k; \ldots; m_n : \tau_n \rangle <: \langle m_1 : \varphi_1; \ldots; m_k : \varphi_k \rangle}$$

$$\frac{(\alpha <: \beta) \in H}{H \vdash \alpha <: \beta} \qquad \frac{H \cup \{(\alpha <: \beta)\} \vdash \tau <: \tau'}{H \vdash \mu\alpha.\ \tau <: \mu\beta.\ \tau'}$$

Proposition 7.1 *La relation $H \vdash \_ <: \_$ est transitive.*

Règles de typage: les règles de typage sont celles de mini-ML monomorphe (section 1.3.2), plus une règle pour les objets, une pour les appels de méthodes, et une règle de subsomption implicite.

$$E \vdash x : E(x) \ \text{(var)} \qquad E \vdash c : TC(c) \ \text{(const)} \qquad E \vdash op : TC(op) \ \text{(op)}$$

$$\frac{E + \{x : \tau_1\} \vdash a : \tau_2}{E \vdash (\texttt{fun}\ x \to a) : \tau_1 \to \tau_2} \ \text{(fun)} \qquad \frac{E \vdash a_1 : \tau' \to \tau \quad E \vdash a_2 : \tau'}{E \vdash a_1\ a_2 : \tau} \ \text{(app)}$$

$$\frac{E \vdash a_1 : \tau_1 \quad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2} \ \text{(paire)} \qquad \frac{E \vdash a_1 : \tau_1 \quad E + \{x : \tau_1\} \vdash a_2 : \tau_2}{E \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \tau_2} \ \text{(let)}$$

$$\frac{i \neq j \Rightarrow m_i \neq m_j \quad E^* \vdash a_i : \tau_i \quad E^* + \{x : \langle m_1 : \tau'_1;\ \ldots;\ m_k : \tau'_k\rangle;\ \texttt{val}\ x_i : \tau_i\} \vdash a'_j : \tau'_j}{E \vdash \texttt{obj}(x)\langle \ldots; \texttt{val}\ x_i = a_i; \ldots; \texttt{method}\ m_j = a'_j; \ldots\rangle : \langle m_1 : \tau'_1;\ \ldots;\ m_k : \tau'_k\rangle} \ \text{(objet)}$$

$$\frac{E \vdash a : \langle m : \tau\rangle}{E \vdash a\#m : \tau} \ \text{(meth)} \qquad \frac{E \vdash a : \tau \quad \emptyset \vdash \tau <: \tau'}{E \vdash a : \tau'} \ \text{(sub)}$$

7.5.2 Construction du système de contraintes

Comme pour l'inférence de types de ML monomorphe (section 3.2), la première phase de l'inférence de types est d'associer au programme un système de contraintes entre types qui caractérise tous les typages possibles pour le programme. Dans la section 3.2, ces contraintes étaient des contraintes d'égalité; ici, ce sont des contraintes de sous-typage.

On se donne un programme (une expression close) a_0 dans laquelle tous les identificateurs liés par `fun` ou `let` ont des noms différents. À chaque identificateur x dans a_0, on associe une variable de type α_x. De même, à chaque sous-expression a de a_0, on associe une variable de type α_a.

Le système d'équations $C(a_0)$ associé à a_0 est construit en parcourant l'expression a_0 et en ajoutant des équations pour chaque sous-expression a de a_0, comme suit:

- Si a est une variable x: $C(a) = \{\alpha_x \overset{?}{<:} \alpha_a\}$.
- Si a est une constante c ou un opérateur op:
  $C(a) = \{TC(c) \overset{?}{<:} \alpha_a\}$ ou $C(a) = \{TC(op) \overset{?}{<:} \alpha_a\}$.
- Si a est $\texttt{fun}\ x \to b$: $C(a) = \{\alpha_x \to \alpha_b \overset{?}{<:} \alpha_a\} \cup C(b)$.
- Si a est une application $b\ c$: $C(a) = \{\alpha_b \overset{?}{<:} \alpha_c \to \alpha_a\} \cup C(b) \cup C(c)$.
- Si a est une paire (b, c): $C(a) = \{\alpha_b \times \alpha_c \overset{?}{<:} \alpha_a\} \cup C(b) \cup C(c)$.
- Si a est $\texttt{let}\ x = b\ \texttt{in}\ c$: $C(a) = \{\alpha_b \overset{?}{<:} \alpha_x;\ \alpha_c \overset{?}{<:} \alpha_a\} \cup C(b) \cup C(c)$.
- Si a est $\texttt{obj}(x)\langle \ldots; \texttt{val}\ x_i = a_i; \ldots; \texttt{method}\ m_j = b_j; \ldots\rangle$:
  $C(a) = \{\langle m_j : \alpha_{b_j}\rangle \overset{?}{<:} \alpha_a;\ \langle m_j : \alpha_{b_j}\rangle \overset{?}{<:} \alpha_x;\ \alpha_{a_i} \overset{?}{<:} \alpha_{x_i}\} \cup C(a_i) \cup C(b_j)$.

- Si a est $b\#m$: $C(a) = \{\alpha_b \overset{?}{<:} \langle m : \alpha_a \rangle\} \cup C(b)$.

Remarquons que $C(a)$ est un ensemble d'inéquations entre types finis (il ne contient pas de types récursifs $\mu\alpha.\ \tau$).

Exemple: on considère le programme

$$a = \mathtt{fun}\ x \to \underbrace{(\underbrace{\underbrace{x}_{d}\#m}_{c}, \underbrace{\underbrace{x}_{f}\#m'}_{e})}_{b}$$

On a:

$$\begin{aligned} C(a) = \{\ & \alpha_x \to \alpha_b \overset{?}{<:} \alpha_a; \\ & \alpha_c \times \alpha_e \overset{?}{<:} \alpha_b; \\ & \alpha_d \overset{?}{<:} \langle m : \alpha_c \rangle; \\ & \alpha_x \overset{?}{<:} \alpha_d; \\ & \alpha_f \overset{?}{<:} \langle m' : \alpha_e \rangle; \\ & \alpha_x \overset{?}{<:} \alpha_f \} \end{aligned}$$

7.5.3 Lien entre typages et solutions des équations

Une solution de l'ensemble de contraintes $C(a)$ est une substitution φ (des variables de types dans les types finis ou infinis) telle que pour toute inéquation $\tau_1 \overset{?}{<:} \tau_2$ dans $C(a)$, on ait $\varphi(\tau_1) <: \varphi(\tau_2)$. Le deux propositions suivantes montrent que les solutions de $C(a)$ caractérisent exactement les typages possibles pour a.

Proposition 7.2 (Correction des solutions vis-à-vis du typage) *Si φ est une solution de $C(a)$, alors $E \vdash a : \varphi(\alpha_a)$ où E est l'environnement de typage $\{x : \varphi(\alpha_x) \mid x \text{ libre dans } a\}$.*

Proposition 7.3 (Complétude des solutions vis-à-vis du typage) *Soit a une expression. S'il existe un environnement E et un type τ tels que $E \vdash a : \tau$, alors le système d'équations $C(a)$ admet une solution.*

7.5.4 Cohérence d'un système de contraintes

Dans la section 3.2, nous étins capables de résoudre le système d'équations $C(a)$ par unification et d'en fournir une solution principale (qui résume toutes les solutions possibles). Ce n'est plus possible avec les systèmes d'inéquations de sous-typage.

Exemple: $\mathtt{let}\ f = \mathtt{fun}\ x \to x\ \mathtt{in}\ f(\mathtt{obj}(s)\langle\mathtt{method}\ m = 1\rangle)$. On peut attribuer à `f` les types $\langle \mathtt{m} : \mathtt{int} \rangle \to \langle \mathtt{m} : \mathtt{int} \rangle$ et $\langle\,\rangle \to \langle\,\rangle$ (entre autres). Cependant, aucun de ces deux types n'est plus général que l'autre. En particulier, ils sont incomparables par la relation de sous-typage.

Au lieu d'essayer de résoudre l'ensemble de contraintes $C(a)$, nous allons simplement établir qu'il existe une solution de $C(a)$ sans la calculer entièrement. Cela suffit à garantir que le programme a est bien typé et s'exécute sans erreurs.

On établit la solvabilité (l'existence d'une solution) d'un ensemble de contraintes C en deux temps: d'abord on calcule la fermeture C^* de C; ensuite, on vérifie que C^* ne contient pas d'incohérences immédiates.

Fermeture: on dit qu'un ensemble de contraintes C entre types finis est fermé (par transitivité et par propagation) s'il satisfait les conditions suivantes:

- Si $\tau_1 \overset{?}{<:} \tau_2 \in C$ et $\tau_2 \overset{?}{<:} \tau_3 \in C$, alors $\tau_1 \overset{?}{<:} \tau_3 \in C$.
- Si $(\tau_1 \to \tau_2) \overset{?}{<:} (\varphi_1 \to \varphi_2) \in C$, alors $\varphi_1 \overset{?}{<:} \tau_1 \in C$ et $\tau_2 \overset{?}{<:} \varphi_2 \in C$.
- Si $(\tau_1 \times \tau_2) \overset{?}{<:} (\varphi_1 \times \varphi_2) \in C$, alors $\tau_1 \overset{?}{<:} \varphi_1 \in C$ et $\tau_2 \overset{?}{<:} \varphi_2 \in C$.
- Si $\langle m_i : \tau_i \rangle \overset{?}{<:} \langle n_j : \varphi_j \rangle \in C$, alors $\tau_i <: \varphi_j \in C$ pour tous i, j tels que $m_i = n_j$ (i.e. pour tous les noms de méthodes qui apparaissent à la fois dans les deux types objet).

On note C^* la fermeture de C, c'est-à-dire le plus petit ensemble fermé contenant C. On l'obtient à partir de C en ajoutant à C toutes les contraintes nécessaires pour satisfaire les conditions ci-dessus. Par exemple, si $\tau_1 \overset{?}{<:} \tau_2 \in C$ et $\tau_2 \overset{?}{<:} \tau_3 \in C$ mais $\tau_1 \overset{?}{<:} \tau_3 \notin C$, on ajoute $\tau_1 \overset{?}{<:} \tau_3$ à C. Le processus termine forcément, car les contraintes ajoutées sont des contraintes entre types qui sont des sous-termes de types apparaissant dans l'ensemble initial C, et ces sous-termes sont en nombre fini.

Proposition 7.4 *Un ensemble de contraintes C est solvable si et seulement si sa fermeture C^* est solvable*

Démonstration: Si C^* est solvable, comme $C \subseteq C^*$, toute solution de C^* est aussi solution de C, donc C est solvable. Réciproquement, toute solution de C satisfait également les inéquations de C^*, car celles-ci sont des conséquences d'inéquations contenues dans C. Par exemple, si $\tau_1 \overset{?}{<:} \tau_3 \in C^*$ avec $\tau_1 \overset{?}{<:} \tau_2 \in C$ et $\tau_2 \overset{?}{<:} \tau_3 \in C$ (transitivité), si une substitution φ est telle que $\varphi(\tau_1) <: \varphi(\tau_2)$ et $\varphi(\tau_2) <: \varphi(\tau_3)$, alors par transitivité de $<:$ on a $\varphi(\tau_1) <: \varphi(\tau_3)$. □

Cohérence immédiate: un ensemble de contraintes C est immédiatement cohérent si toutes les contraintes $\tau_1 \overset{?}{<:} \tau_2$ dans C tombent dans l'un des cas suivants:

- τ_1 ou τ_2 sont des variables de types;
- $\tau_1 = \perp$;
- $\tau_2 = \top$;
- τ_1 et τ_2 sont des types flèches;
- τ_1 et τ_2 sont des types produits;

- $\tau_1 = \langle m_i : \varphi_i \rangle$, $\tau_2 = \langle n_j : \psi_j \rangle$, et l'ensemble de noms de méthodes $\{n_j\}$ est inclus dans l'ensemble des noms de méthodes $\{m_i\}$.

Un ensemble qui n'est pas immédiatement cohérent est dit immédiatement incohérent.

Proposition 7.5 *Un ensemble de contraintes immédiatement incohérent n'est pas solvable.*

Démonstration: si $\tau_1 \stackrel{?}{<:} \tau_2$ est une équation incohérente de C, pour toute substitution φ, on ne peut pas dériver $H \vdash \varphi(\tau_1) <: \varphi(\tau_2)$ car cet énoncé n'a la forme d'aucune conclusion d'une des règles d'inférence définissant $<:$. □

La réciproque de la proposition précédente n'est pas vraie si C n'est pas fermé. Par exemple, $\{\mathtt{int} \to \mathtt{int} \stackrel{?}{<:} \alpha;\ \alpha \stackrel{?}{<:} \mathtt{int} \times \mathtt{int}\}$ est immédiatement cohérent, mais n'admet pas de solutions. Cependant, si C est fermé, nous avons le théorème suivant (que nous admettrons):

Proposition 7.6 *Un ensemble de contraintes C fermé et immédiatement cohérent est solvable.*

7.5.5 Algorithme d'inférence de types

En combinant les résultats précédents, nous obtenons l'algorithme d'inférence de types $I(a)$ suivant:

Entrée: une expression close a.
Sortie: "bien typé" ou "non typable".
Calcul: si $C(a)^*$ est immédiatement incohérent, renvoyer "non typable". Sinon, renvoyer "bien typé".

Proposition 7.7 (Correction et complétude de l'inférence) $I(a)$ *renvoie "bien typé" si et seulement si il existe τ tel que $\emptyset \vdash a : \tau$.*

Démonstration: la partie "si" découle des propositions 7.2, 7.4 et 7.5. La partie "seulement si" est corollaire des propositions 7.3, 7.4 et 7.6. □

Lectures pour aller plus loin

- Martín Abadi et Luca Cardelli, *A theory of objects*, Springer-Verlag Monographs in Computer Science, 1996.
  Un livre entier sur les calculs d'objets et leurs systèmes de types (sans inférence).

- Didier Rémy et Jérôme Vouillon, *Objective ML: An effective object-oriented extension to ML*, Theory And Practice of Object Systems, 4(1):27–50, 1998, `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/objective-ml!tapos98.ps.gz`
  L'approche Objective Caml.

- Roberto Amadio et Luca Cardelli, *Subtyping recursive types*, ACM Transactions on Programming Languages and Systems, 15(4), 1993, `http://research.microsoft.com/Users/luca/Papers/SRT.A4.ps`
  La référence sur le sous-typage entre types récursifs.

Exercices

Exercice 7.1 *(*)/(**) Montrer que la relation de sous-typage (sans types récursifs) est une relation d'ordre: réflexive ($\tau <: \tau$), transitive ($\tau_1 <: \tau_2$ et $\tau_2 <: \tau_3$ implique $\tau_1 <: \tau_3$) et antisymétrique ($\tau <: \tau'$ et $\tau' <: \tau$ impliquent $\tau = \tau'$).*

Exercice de programmation 7.1 *Implémenter le prédicat <: de la section 7.2. (Indication: attention à l'équation de commutativité sur les rangées.)*

Exercice 7.2 *(**)/(***) Montrer le cas β_{fun} de la préservation du typage par réduction de tête (l'analogue de la proposition 2.3) lorsqu'on ajoute la règle (sub). (Indication: attention, il y a un piège.)*

Exercice 7.3 *(*) Montrer que les types récursifs permettent de typer en ML tous les termes du λ-calcul pur. Quel est le type qu'ont tous les λ-termes?*

Exercice 7.4 *(**)/(***) Le but de cet exercice est de comprendre la représentation des types récursifs par des graphes et de programmer l'algorithme d'unification correspondant.*

Un graphe de types est un graphe orienté. Chaque noeud est étiqueté ou bien par le symbole V (signifiant que ce noeud représente une variable de type), ou bien par un constructeur de type comme `int`, `bool`, $\to$, $\times$. (Pour simplifier, on ne considère pas les types objets.) Chaque noeud doit avoir un nombre de noeuds fils égal à l'arité $A(c)$ de son étiquette c, avec bien sûr $A(V) = A(\texttt{int}) = A(\texttt{bool}) = A(\emptyset) = 0$ (pas de fils), et $A(\to) = A(\times) = 2$ (deux fils). Un noeud du graphe représente donc une expression de type dont le constructeur de tête est donné par l'étiquette du type et dont les sous-expressions sont représentées par les fils éventuels de ce noeud. Les types récursifs apparaissent naturellement sous forme de cycles dans le graphe.

Si n est un noeud du graphe, on note $C(n)$ son constructeur de type et $F_i(n)$ le noeud du i^e fils de n (avec $1 \leq i \leq A(C(n))$).

Une substitution, dans ce formalisme, est une relation d'équivalence R entre les noeuds du graphe qui vérifie les propriétés suivantes:

- *(Compatibilité des constructeurs) Si $n\ R\ n'$ et $C(n) \neq V$ et $C(n') \neq V$, alors $C(n) = C(n')$. (Autrement dit, une subsitution ne peut rendre égaux que des noeuds qui ont le même constructeur, sauf si l'un des deux est une variable; un noeud variable peut être rendu égal à n'importe quel noeud.)*
- *(Fermeture) Si $n\ R\ n'$ et $C(n) \neq V$ et $C(n') \neq V$, alors $F_i(n)\ R\ F_i(n')$ pour $1 \leq i \leq A(C(n))$. (Autrement dit, une substitution qui égalise deux noeuds non variables doit aussi égaliser leurs fils deux à deux.)*

Un ensemble d'équations sur un graphe de types est un ensemble de paires de noeuds $\{n \stackrel{?}{=} n'; n_1 \stackrel{?}{=} n'_1; \ldots\}$. Un unificateur d'un ensemble d'équations E est une substitution R telle que $n\ R\ n'$ pour tout $(n \stackrel{?}{=} n') \in E$. L'unificateur R est principal si tout autre unificateur R' est une relation moins fine que R, c'est-à-dire $n_1\ R\ n_2 \Rightarrow n_1\ R'\ n_2$.

L'algorithme suivant calcule l'unificateur principal d'un ensemble d'équations E. Il prend en entrée une relation d'équivalence R qui représente des identifications entre noeuds déjà effectuées, *et retourne une relation d'équivalence qui est l'unificateur principal.*

$$\begin{array}{rcl}
\mathtt{mgu}(\emptyset, R) & = & R \\
\mathtt{mgu}(\{n \stackrel{?}{=} n'\} \cup E, R) & = & \mathtt{mgu}(E, R) \textit{ si } n\ R\ n' \\
\mathtt{mgu}(\{n \stackrel{?}{=} n'\} \cup E, R) & = & \mathtt{mgu}(E, R + \{n = n'\}) \textit{ si } C(n) = V \textit{ ou } C(n') = V \\
\mathtt{mgu}(\{n \stackrel{?}{=} n'\} \cup E, R) & = & \mathtt{mgu}(E \cup \{F_1(n) \stackrel{?}{=} F_1(n'); \dots; F_k(n) \stackrel{?}{=} F_k(n')\}, R + \{n = n'\}) \\
 & & \textit{si } C(n) \neq V \textit{ et } C(n') = C(n) \textit{ et } k = A(C(n)) \\
\mathtt{mgu}(\{n \stackrel{?}{=} n'\} \cup E, R) & = & \textit{échec si } C(n) \neq V \textit{ et } C(n') \neq C(n)
\end{array}$$

On a noté $R + \{n = n'\}$ la plus fine relation d'équivalence qui contient R et qui relie n et n'. Elle s'obtient à partir de R en fusionnant les classes d'équivalence de n et n' dans R.

1) Dessiner les graphes représentant les types suivants:
$\mathtt{int} \to \mathtt{bool}$; $\alpha \times \beta$; $\alpha \times \alpha$; $\mu\alpha.\ \mathtt{int} \to \alpha$; $\mu\alpha.\ \beta \to \gamma \to \alpha$.

2) Faire tourner à la main l'algorithme sur la représentation sous forme de graphe du problème d'unification $\mu\alpha.\ \mathtt{int} \to \alpha \stackrel{?}{=} \mu\alpha.\ \beta \to \gamma \to \alpha$. (On partira de la relation identité comme second paramètre initial de `mgu`*.) Quelle est la forme de la substitution renvoyée?*

3) Montrer que si $R = \mathtt{mgu}(E, id)$ (où id est la relation identité) est définie, alors R est une substitution. Montrer que R est un unificateur de E. Montrer que R est un unificateur principal de E.

4) Montrer que l'algorithme `mgu` *termine toujours.*

Exercice de programmation 7.2 *Implémenter l'algorithme d'unification de graphes de l'exercice précédent. (Indication: on représentera la relation d'équivalence entre noeuds par une structure de type* union-find*. C'est-à-dire, on placera dans chaque noeud un champ mutable de type* `noeud option`*, dont la signification est la suivante: si* `None`*, cela veut dire que le noeud n'est encore pas identifié à un autre noeud; si* `Some`*(n), cela veut dire que le noeud est dans la même classe d'équivalence que le noeud n. Au lieu de renvoyer une relation d'équivalence comme résultat de l'unification, on modifiera en place ces champs mutables types pour représenter cette relation.)*

Exercice 7.5 *(***, pour mathématiciens) Le but de cet exercice est de justifier mathématiquement l'existence des types $\mu\alpha.\ \tau$. L'idée est de construire les types infinis comme limites de suites convergentes de types finis, de même que Cantor construisit les réels comme limites de suites convergentes de rationnels.*

1) Pour simplifier, on considère uniquement l'algèbre de types $\tau ::= \alpha \mid \tau_1 \to \tau_2$. On définit la distance $d(\tau_1, \tau_2)$ entre deux types (finis) τ_1 et τ_2 de la façon suivante:

$$\begin{array}{rcl}
d(\tau_1 \to \varphi_1, \tau_2 \to \varphi_2) & = & \frac{1}{2} \max(d(\tau_1, \tau_2), d(\varphi_1, \varphi_2)) \\
d(\alpha, \alpha) & = & 0 \\
d(\alpha, \beta) & = & 1 \textit{ si } \alpha \neq \beta \\
d(\alpha, \tau_2 \to \varphi_2) & = & 1 \\
d(\tau_1 \to \varphi_1, \beta) & = & 1
\end{array}$$

Montrer que d est une distance ultramétrique, c'est-à-dire qu'elle satisfait l'inégalité du triangle ultramétrique $d(\tau_1, \tau_3) \leq \max(d(\tau_1, \tau_2), d(\tau_2, \tau_3))$, qu'elle est symétrique, et que de plus $d(\tau_1, \tau_2) = 0$ si et seulement si $\tau_1 = \tau_2$.

2) (Complétion d'un espace métrique.) On considère l'ensemble $\mathcal{S}$ dont les éléments sont des suites de Cauchy de types $(\tau_n)_{n\in\mathbf{N}}$. On rappelle qu'une suite (τ_n) est de Cauchy si

$$\forall\varepsilon.\exists n.\forall p,q \geq n.\ d(\tau_p,\tau_q) \leq \varepsilon.$$

On définit la distance $d(s,s')$ entre deux telles suites $s=(\tau_n)$ et $s'=(\tau'_n)$ par

$$d(s,s') = \lim_{n\to\infty} d(\tau_n,\tau'_n).$$

a) Montrer que cette limite existe toujours. b) Montrer que la relation entre suites $\cong$ définie par $s \cong s' \Leftrightarrow d(s,s') = 0$ est une relation d'équivalence. c) Montrer que l'ensemble quotient $\mathcal{T} = \mathcal{S}/\cong$ muni de la distance $d/\cong$ est un espace ultramétrique. d) Montrer que les types simples se plongent naturellement dans $\mathcal{T}$ et que les distances sont préservées par le plongement. e) Montrer ou admettre que $\mathcal{T}$ est complet (toute suite de Cauchy d'éléments de $\mathcal{T}$ converge vers un élement de $\mathcal{T}$).

3) (Théorème de Banach-Tarski.) Soit (E,d) un espace métrique complet. Une fonction F de E dans E est dite contractive s'il existe une constante $k \in]0,1[$ telle que $d(F(x),F(y)) \leq k\,d(x,y)$ pour tous $x,y \in E$. Montrer que toute fonction contractive admet un point fixe et que ce point fixe est unique.

4) Soit α une variable de type et $\tau \in \mathcal{T}$ tel que $\tau \neq \alpha$. Montrer qu'il existe un et un seul $\tau' \in \mathcal{T}$ tel que $\tau' = \tau[\alpha \leftarrow \tau']$.