Types et contraintes

François Pottier

15 décembre 2004

Types

Un *type* est une description concise du comportement d'un fragment de programme.

Le typage peut fournir des garanties de *sûreté* ou de *sécurité*.

Il favorise également la *modularité* et *l'abstraction*.

Contraintes

Un fragment de programme admet un type si et seulement si chacun des sous-fragments qui le composent admet lui-même un type, et si ces types sont cohérents les uns vis-à-vis des autres.

Un problème *d'inférence de types* s'exprime donc typiquement sous forme d'une *contrainte* composée de prédicats sur les types, de conjonctions, et de quantifications existentielles.

De plus, les contraintes peuvent participer à la *spécification* même d'un système de types.

Spécification

La spécification classique du système de types de Damas et Milner est un jeu de règles permettant de dériver des *jugements*:

$$\Gamma \vdash x : \Gamma(x) \qquad \frac{\Gamma; x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \longrightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \longrightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma; x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad \bar{a} \# \text{ftv}(\Gamma)}{\Gamma \vdash e : \forall \bar{a}.\tau} \qquad \frac{\Gamma \vdash e : \forall \bar{a}.\tau}{\Gamma \vdash e : [\vec{\tau}/\vec{a}]\tau}$$

L'exécution d'un programme bien typé ne peut échouer.

L'algorithme $\mathcal{J}$ de Milner (extrait)

L'algorithme attend un pré-jugement $\Gamma \vdash e$, produit un type τ, et emploie deux variables globales V et ϕ.

$$
\begin{array}{lcl}
\mathcal{J}(\Gamma \vdash e_1\, e_2) & = & \text{do } \tau_1 = \mathcal{J}(\Gamma \vdash e_1) \\
& & \text{do } \tau_2 = \mathcal{J}(\Gamma \vdash e_2) \\
& & \text{do } a = \text{fresh} \\
& & \text{do } \phi \leftarrow \text{mgu}(\phi(\tau_1) = \phi(\tau_2 \to a)) \circ \phi \\
& & \text{return } a \\
\mathcal{J}(\Gamma \vdash \text{let } x = e_1 \text{ in } e_2) & = & \text{do } \tau_1 = \mathcal{J}(\Gamma \vdash e_1) \\
& & \text{let } \sigma = \overline{\forall}\, \text{ftv}(\phi(\Gamma)).\phi(\tau_1) \\
& & \text{return } \mathcal{J}(\Gamma; x : \sigma \vdash e_2)
\end{array}
$$

Production et *résolution* d'équations sont entremêlées.

Correction et complétude de l'algorithme $\mathcal{J}$

Théorème (Correction)
Si $\mathcal{J}(\Gamma \vdash e)$ termine dans l'état (ϕ, V) et renvoie τ, alors $\phi(\Gamma) \vdash e : \phi(\tau)$ est un jugement.

Théorème (Complétude)
Soit Γ un environnement de typage. Soit (ϕ_0, V_0) un état satisfaisant les invariants de l'algorithme. Supposons donnés θ_0 et τ_0 tels que $\theta_0\phi_0(\Gamma) \vdash e : \tau_0$ soit un jugement. Alors, l'exécution de $\mathcal{J}(\Gamma \vdash e)$ à partir de l'état initial (ϕ_0, V_0) réussit. Soient (ϕ_1, V_1) son état final et τ_1 le type renvoyé. Alors il existe une substitution θ_1 telle que $\theta_0\phi_0$ et $\theta_1\phi_1$ coïncident en dehors de V_0 et telle que τ_0 s'écrit $\theta_1\phi_1(\tau_1)$.

Complétude de l'algorithme $\mathcal{J}$ (extrait de la preuve)

[...] Nous avons

$$\theta_1\phi_1(\gamma) = \theta_1\psi\phi'_2(\gamma) = \theta''_2\phi'_2(\gamma).$$

Puisque a est distinct de γ et hors de portée de ϕ'_2, nous pouvons continuer avec

$$\theta''_2\phi'_2(\gamma) = \theta'_2\phi'_2(\gamma) = \theta'_1\phi'_1(\gamma) = \theta_0\phi_0(\gamma).$$

$\theta_1\phi_1$ et $\theta_0\phi_0$ coïncident donc en dehors de V_0 [...]

Syntaxe

La syntaxe des *contraintes* et des *schémas de types contraints* sera la suivante:

$$\begin{array}{lcl} C & ::= & \tau \leq \tau \mid C \wedge C \mid \exists a.C \mid x \preceq \tau \mid \mathsf{def}\ x : \sigma\ \mathsf{in}\ C \\ \sigma & ::= & \forall \bar{a}[C].\tau \end{array}$$

Interprétation

Les contraintes sont interprétées dans un *modèle* logique.

Une variable a dénote un élément d'un univers de Herbrand.

Une variable x dénote un ensemble de tels éléments.

L'interprétation logique des contraintes est définie de façon à satisfaire les lois

$$\text{def } x : \sigma \text{ in } C \equiv [\sigma/x]C$$

$$(\forall \bar{a}[C].\tau) \preceq \tau' \equiv \exists \bar{a}.(C \wedge \tau \leq \tau')$$

Production

À une expression e et un type τ, on associe une contrainte $[\![e : \tau]\!]$, nécessaire et suffisante pour que e admette le type τ.

$$
\begin{array}{rcl}
[\![x : \tau]\!] & = & x \preceq \tau \\
[\![\lambda x.e : \tau]\!] & = & \exists a_1 a_2. \left(\begin{array}{l} \text{def } x : a_1 \text{ in } [\![e : a_2]\!] \\ a_1 \to a_2 \leq \tau \end{array} \right) \\
[\![e_1\, e_2 : \tau]\!] & = & \exists a.([\![e_1 : a \to \tau]\!] \wedge [\![e_2 : a]\!]) \\
[\![\text{let } x = e_1 \text{ in } e_2 : \tau]\!] & = & \text{let } x : \forall a[[\![e_1 : a]\!]].a \text{ in } [\![e_2 : \tau]\!]
\end{array}
$$

Cette spécification est *équivalente* aux précédentes.

Spécification

Les jugements font explicitement apparaître une *contrainte*:

$$\frac{\Gamma(x) = \sigma \qquad C \Vdash \exists \sigma}{C, \Gamma \vdash x : \sigma} \qquad \frac{C, \Gamma; x : \tau_1 \vdash e : \tau_2}{C, \Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{C, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad C, \Gamma \vdash e_2 : \tau_1}{C, \Gamma \vdash e_1\, e_2 : \tau_2} \qquad \frac{C, \Gamma \vdash e_1 : \sigma \quad C, \Gamma; x : \sigma \vdash e_2 : \tau}{C, \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}$$

Spécification (suite)

Cette contrainte joue le rôle d'hypothèse lorsque l'on souhaite prouver qu'une assertion portant sur les types est satisfaite.

$$\frac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{a} \# \mathrm{ftv}(C, \Gamma)}{C \wedge D, \Gamma \vdash e : \forall \bar{a}[D].\tau} \qquad \frac{C, \Gamma \vdash e : \forall \bar{a}[D].\tau \qquad C \Vdash D}{C, \Gamma \vdash e : \tau}$$

$$\frac{C, \Gamma \vdash e : \tau \qquad C \Vdash \tau \leq \tau'}{C, \Gamma \vdash e : \tau'} \qquad \frac{C, \Gamma \vdash e : \sigma \qquad \bar{a} \# \mathrm{ftv}(\Gamma, \sigma)}{\exists \bar{a}.C, \Gamma \vdash e : \sigma}$$

La présence d'une contrainte dans les jugements conduit naturellement à l'emploi de schémas de types *contraints*.

Les deux facettes de HM(X)

Cette spécification est *équivalente* aux règles de production de contraintes précédentes.

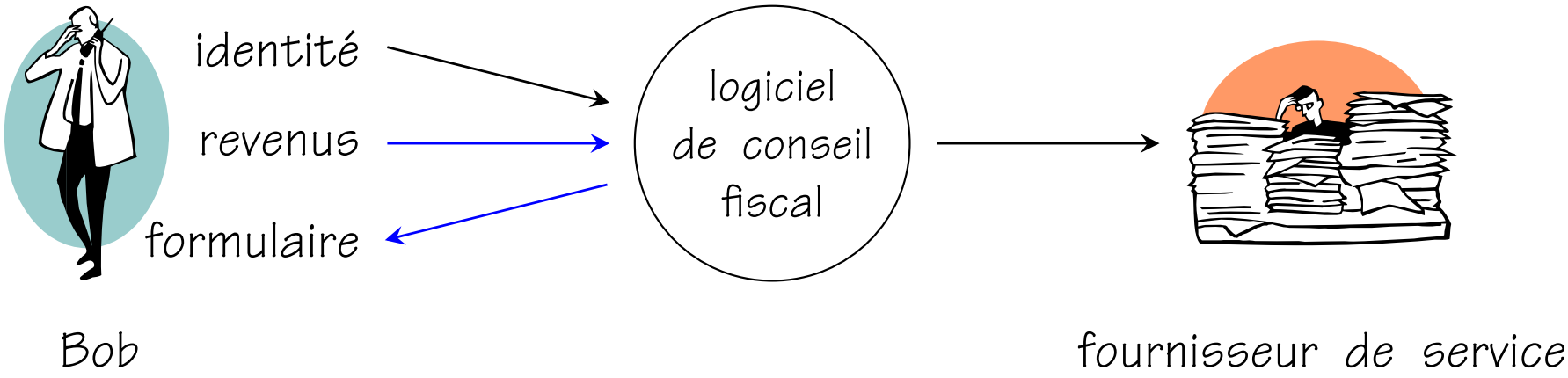
Théorème

$C, \Gamma \vdash e : \tau$ est équivalent à $C \Vdash \mathit{def}\ \Gamma\ \mathit{in}\ [\![e : \tau]\!]$.

Corollaire

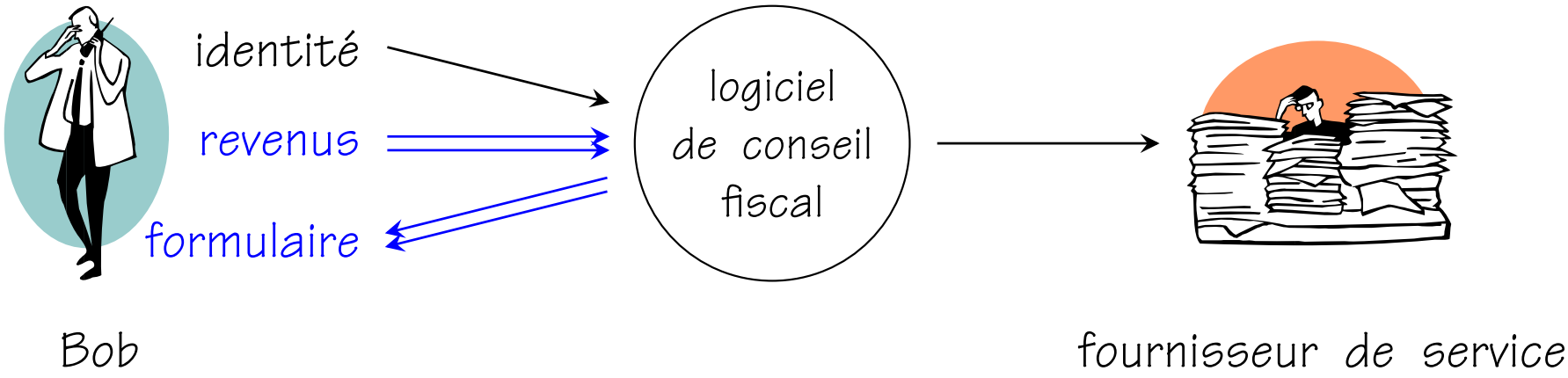
Une expression close e est bien typée si et seulement si la contrainte $\exists a.[\![e : a]\!]$ est satisfiable.

Pourquoi contrôler les flots d'information ?



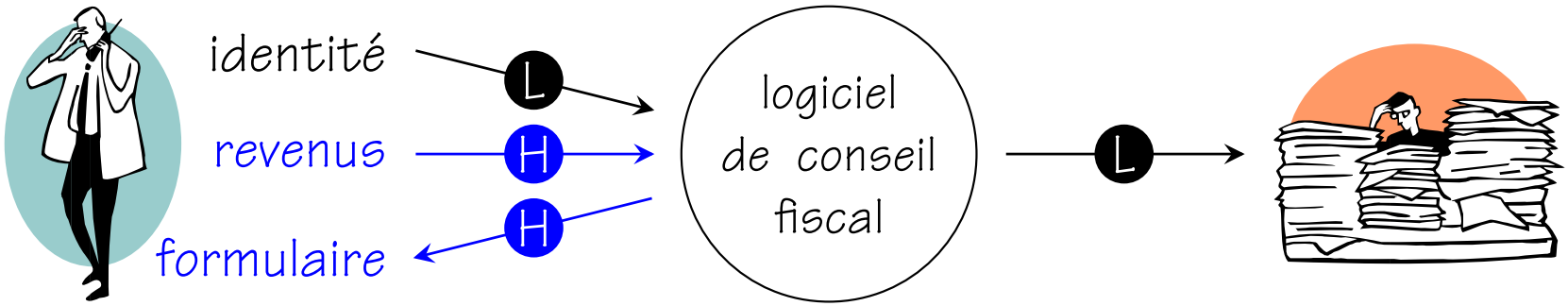
Bob emploie un logiciel pour remplir son formulaire de déclaration de revenus, lequel communique, à travers le réseau, avec son site d'origine.

Pourquoi contrôler les flots d'information ?



Bob souhaiterait obtenir une garantie de *secret* : s'il modifiait les revenus qu'il déclare, le fournisseur du logiciel ne pourrait percevoir aucune différence. En d'autres termes, les données envoyées au fournisseur ne *dépendent* pas des revenus de Bob.

Pourquoi contrôler les flots d'information ?

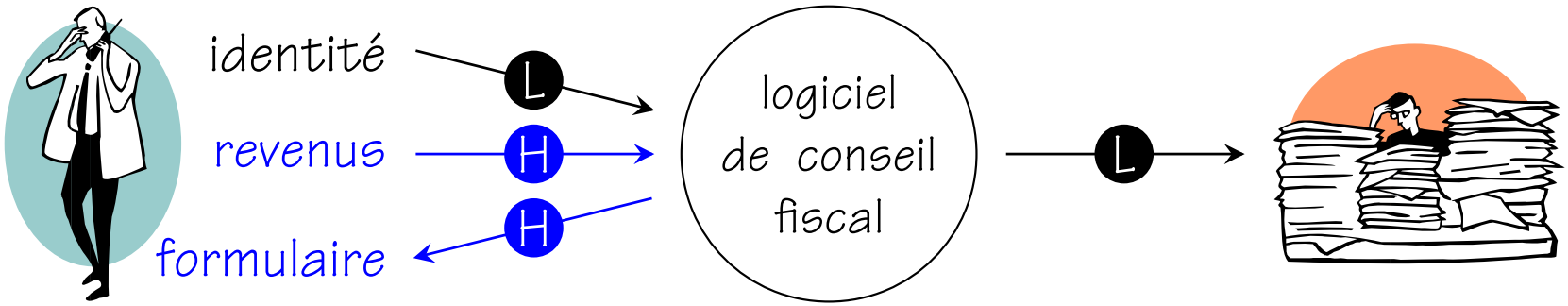


Bob

fournisseur de service

Pour spécifier cela de façon succincte, on attribue des *niveaux* d'information ordonnés à chaque canal d'entrée ou de sortie — ici, L < H — et on n'autorise la transmission d'information que vers le *haut*.

Pourquoi contrôler les flots d'information ?



Bob

fournisseur de service

Cette spécification peut prendre la forme d'un *type*, comme

$$string^{L} \times int^{H} \longrightarrow string^{H} \times int^{L}$$

Polymorphisme

Pour permettre la réutilisation de code, le système est doté de polymorphisme vis-à-vis des *niveaux*, des *types*, et des effets:

$$\begin{array}{rcl} \lambda x.x + 1 & : & \forall a.int^a \longrightarrow int^a \\ \lambda x.x & : & \forall \beta.\beta \longrightarrow \beta \end{array}$$

Le polymorphisme permet également d'*abstraire* une spécification vis-à-vis du treillis des niveaux d'information:

$$\begin{array}{r} string^{\mathsf{L}} \times int^{\mathsf{H}} \longrightarrow string^{\mathsf{H}} \times int^{\mathsf{L}} \\ \forall a\beta[a \leq \beta].string^a \times int^\beta \longrightarrow string^\beta \times int^a \end{array}$$

Sous-typage

Les types habituels sont décorés par des niveaux, comme L et H. La relation d'ordre entre niveaux donne donc naturellement lieu à une relation de *sous-typage structurel* entre types.

Le sous-typage permet une analyse *orientée* des flots d'information:

$$\lambda(a,b,c).(b+c,a+c,a+b) :$$

$$\forall \begin{bmatrix} \beta \leq a' & \gamma \leq a' \\ a \leq \beta' & \gamma \leq \beta' \\ a \leq \gamma' & \beta \leq \gamma' \end{bmatrix} . int^a \times int^\beta \times int^\gamma \longrightarrow int^{a'} \times int^{\beta'} \times int^{\gamma'}$$

Contraintes non standard

Le système emploie également des contraintes d'ordre entre un *niveau* et un *type*:

$$\begin{array}{l}\text{let } \mathit{choose}\ b\ x\ y = \text{if } b \text{ then } x \text{ else } y \\ \forall a\beta[a \leq \mathit{level}(\beta)].\mathit{bool}^a \to \beta \to \beta \to \beta\end{array}$$

Une telle contrainte exige, en gros, que "l'étiquette" portée par le type β soit bornée inférieurement par le niveau a.

Contraintes non standard (suite)

On dispose également de contraintes d'ordre entre un *type* et un *niveau*:

$$(=) : \forall a\beta[\mathit{content}(a) \leq \beta].a \longrightarrow a \longrightarrow \mathit{bool}^{\beta}$$

Une telle contrainte exige, en gros, que le niveau β soit borné inférieurement par "toutes les étiquettes" du type a.

Rangées

Tout constructeur flèche est annoté par une *rangée* qui à chaque nom d'exception associe le niveau de l'information trahie par le lancement de cette exception.

$$\lambda x.(\text{if } x \text{ then raise } X)$$

$$\forall \alpha\beta\gamma[\alpha \leq \gamma \wedge \beta \leq \gamma].\mathit{bool}^{\alpha} \xrightarrow{\beta \quad X:\gamma} \mathit{unit}$$

Spécification

L'analyse est spécifiée par un jeu de règles dont le style rappelle la spécification de HM(X). Par exemple,

$$\frac{C, \Gamma \vdash v : bool^{\lambda} \qquad \forall j \in \{1,2\} \quad C, \pi \sqcup \lambda, \Gamma \vdash e_j : \tau\ [\rho] \qquad C \Vdash \lambda \leq level(\tau)}{C, \pi, \Gamma \vdash \text{if } v \text{ then } e_1 \text{ else } e_2 : \tau\ [\rho]}$$

Inférence de types

L'inférence de types, donc l'analyse *automatique* de flots d'information, se ramène comme précédemment à la résolution de contraintes. Par exemple,

$$[\![\pi \vdash \text{if } v \text{ then } e_1 \text{ else } e_2 : \tau\ [\rho]]\!] = \exists \alpha\beta. \left(\begin{array}{l} [\![v : bool^a]\!] \\ \pi \leq \beta \wedge a \leq \beta \\ [\![\beta \vdash e_1 : \tau\ [\rho]]\!] \\ [\![\beta \vdash e_2 : \tau\ [\rho]]\!] \\ a \leq level(\tau) \end{array} \right)$$

Résolution de contraintes

Le langage de contraintes considéré comprend une relation de *sous-typage* structurel sur les *types*, *niveaux* et *rangées*, ainsi que *deux relations d'ordre* entre types et niveaux et entre niveaux et types.

La complexité ajoutée par l'analyse vis-à-vis de l'inférence de types classique est bornée par un facteur constant.

Les types algébriques

Les constructeurs de données associés à un type algébrique ordinaire ε reçoivent des schémas de types de la forme:

$$K :: \forall \bar{a}.\tau_1 \cdots \tau_n \longrightarrow \varepsilon(\bar{a})$$

Par exemple,

$$\mathit{Feuille} :: \forall a.\mathit{arbre}(a)$$
$$\mathit{Noeud} :: \forall a.\mathit{arbre}(a) \cdot a \cdot \mathit{arbre}(a) \longrightarrow \mathit{arbre}(a)$$

Filtrer une valeur de type $\mathit{arbre}(a)$ contre le motif $\mathit{Noeud}(l, v, r)$ lie l, v et r à des valeurs de types $\mathit{arbre}(a)$, a et $\mathit{arbre}(a)$.

Les types existentiels à la Läufer & Odersky

Dans l'extension de ML proposée par Läufer et Odersky, les constructeurs de données reçoivent des schémas de types de la forme:

$$K :: \forall \bar{a}\bar{\beta}.\tau_1 \cdots \tau_n \longrightarrow \varepsilon(\bar{a})$$

Par exemple,

$$\mathit{Clef} :: \forall \beta.\beta \cdot (\beta \longrightarrow \mathit{int}) \longrightarrow \mathit{clef}$$

Filtrer une valeur de type *clef* contre le motif $\mathit{Clef}\,(v, f)$ lie v et f à des valeurs de types β et $\beta \longrightarrow \mathit{int}$, *pour un β inconnu*.

Les types algébriques généralisés

Attribuons à présent des schémas de types *contraints* aux constructeurs de données:

$$K :: \forall \bar{a}\bar{\beta}[D].\tau_1 \cdots \tau_n \longrightarrow \varepsilon(\bar{a})$$

Filtrer une valeur de type $\varepsilon(\bar{a})$ contre le motif $K\, x_1 \cdots x_n$ lie x_i à une valeur de type τ_i, *pour des types $\bar{\beta}$ inconnus mais qui satisfont la contrainte D*.

Si D mentionne $\bar{\beta}$, ceux-ci ne sont que *partiellement abstraits*.
Si D mentionne $\bar{a}$, la réussite du filtrage fournit une *information statique supplémentaire* sur ces types existants.

Un exemple typique

Suivant Crary, Weirich et Morrisett, on peut définir un type de *descripteurs de types à l'exécution*:

$$\begin{aligned} \mathit{Entier} &:: \mathit{desc}(\mathit{int}) \\ \mathit{Paire} &:: \forall \beta_1 \beta_2 . \mathit{desc}(\beta_1) \cdot \mathit{desc}(\beta_2) \longrightarrow \mathit{desc}(\beta_1 \times \beta_2) \end{aligned}$$

Ceci peut également s'écrire

$$\begin{aligned} \mathit{Entier} &:: \forall a[a = \mathit{int}].\mathit{desc}(a) \\ \mathit{Paire} &:: \forall a \beta_1 \beta_2 [a = \beta_1 \times \beta_2].\mathit{desc}(\beta_1) \cdot \mathit{desc}(\beta_2) \longrightarrow \mathit{desc}(a) \end{aligned}$$

desc est un constructeur de types *singleton*, au sens où, par exemple, $\mathit{Paire}(\mathit{Entier}, \mathit{Entier})$ est *l'unique* valeur de type $\mathit{desc}(\mathit{int} \times \mathit{int})$.

Un exemple typique (suite)

Ceci permet la définition de fonctions *génériques*:

```
let rec print : ∀a.desc(a) → a → unit = fun t →
  match t with
  | Entier →
      (* a = int *)
      print_int
  | Paire (t1, t2) →
      (* a = β1 × β2 *)
      fun (x1, x2) →
        print t1 x1; print_string " * "; print t2 x2
```

Les deux branches admettent les types *incompatibles* $int \to unit$ et $\beta_1 \times \beta_2 \to unit$, mais *elles ont également un type commun*, à savoir $a \to unit$.

Lien avec les type classes

Dans un langage doté de *type classes*, on aurait pu définir print comme une fonction *surchargée*:

```
class Printable a where
  print :: a → unit
```

Le type de print est alors $\forall a[\text{Printable } a].a \to unit$.

Lien avec les type classes (suite)

On aurait ensuite spécifié que print est applicable aux entiers et aux produits:

```
instance Printable int where
  print = print_int

instance Printable β1, Printable β2 ⇒ Printable β1 × β2 where
  print (x1, x2) = print x1; print_string " * "; print x2
```

Lien avec les type classes (suite)

Il existe une traduction *systématique* de ce programme vers le précédent.

Dans le programme source, le *prédicat* Printable $(int \times int)$ est dérivable. Dans le programme cible, le *type* $desc(int \times int)$ est *habité* par la valeur $Paire(Entier, Entier)$.

Au-delà de l'égalité

On peut imaginer de combiner les types algébriques généralisés avec des langages de contraintes plus riches:

- *arithmétique de Presburger* (Xi)
- *polynômes complexes* (Zenger)
- *sous-typage* (Tse et Zdancewic)
- et autres?

Il semble naturel d'étudier ces combinaisons de façon commune, donc *d'ajouter les types algébriques généralisés à HM(X).*

La facette logique de HMG(X)

La spécification logique de HM(X) se voit ajouter de nouvelles règles, concernant en particulier les fonctions définies par cas:

$$\frac{C \vdash p : \tau' \rightsquigarrow \exists \bar{\beta}[D]\Gamma' \qquad C \wedge D, \Gamma\Gamma' \vdash e : \tau \qquad \bar{\beta} \# \mathrm{ftv}(C, \Gamma, \tau)}{C, \Gamma \vdash p.e : \tau' \longrightarrow \tau}$$

L'exécution d'un programme bien typé ne peut échouer.

La facette fonctionnelle de HMG(X)

Une nouvelle règle s'applique aux fonctions définies par cas:

$$[\![p.e : \tau_1 \longrightarrow \tau_2]\!] = [\![p \downarrow \tau_1]\!] \wedge \forall \bar{\beta}.D \Rightarrow \mathit{def}\ \Gamma'\ \mathit{in}\ [\![e : \tau_2]\!]$$
$$\text{où } \exists \bar{\beta}[D]\Gamma' \text{ est } [\![p \uparrow \tau_1]\!]$$

Les types algébriques généralisés exigent la quantification universelle (déjà requise par les types existentiels à la Läufer & Odersky) et *l'implication*.

Comme dans HM(X), les deux spécifications sont équivalentes.

Un problème ouvert

La contrainte $[\![e : \tau]\!]$ est à présent exprimée (au minimum) dans la *théorie du premier ordre de l'égalité* dans un univers de Herbrand, pour laquelle le problème de la satisfiabilité est de complexité non élémentaire.

Comment restreindre l'ensemble des programmes acceptables, de façon *simple*, *élégante*, *sans perte d'expressivité*, et de sorte à garantir que le problème de résolution de contraintes retrouve une complexité faible ?

Un *credo*

Les contraintes constituent un formalisme *déclaratif* pour *raisonner* et *calculer* à propos des programmes.

Et ensuite ?

L'équipe Cristal veut définir le successeur potentiel d'Objective Caml. Celui-ci devra être plus léger, plus expressif, et pourrait proposer, entre autres:

- plus de polymorphisme ;
- des modules plus maniables et composables ;
- la possibilité d'exprimer des invariants plus riches.