Vier
syntaxe abstraite et règles de typage

Gilles Dubochet

version 1.1

Représenter le programme : syntaxe abstraite

$$
\begin{array}{llcl}
\text{nom référençable} & & & a, b \\
\text{nom de classe} & & & t, u \\
\text{nombre} & n & & \\
\text{programme} & P & ::= & \overline{D}\ e \\
\text{classe} & D & ::= & \texttt{class}\ t \lhd s\ \{\ \overline{d}\ \} \\
 & s & ::= & t \\
 & & | & \texttt{none} \\
\text{membre} & d & ::= & \texttt{field}\ a : T\ =\ e \\
 & & | & \texttt{method}\ a\ (\ \overline{a} : \overline{T}\ ) : T\ =\ e \\
\text{type} & T, U & ::= & t \\
 & & | & \texttt{Int} \\
 & & | & \texttt{Null}
\end{array}
$$

$$
\begin{array}{llcll}
\text{expression} & e, f & ::= & a & \text{paramètre} \\
 & & | & n & \text{nombre} \\
 & & | & \texttt{new}\ t & \text{nouvelle instance} \\
 & & | & \texttt{null} & \text{valeur “}\texttt{null}\text{”} \\
 & & | & e.a & \text{sélection de champ} \\
 & & | & e.a(\overline{e}) & \text{appel de méthode} \\
 & & | & e.a\ =\ e & \text{assignation de champ} \\
 & & | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e & \text{condition “}\texttt{if}\text{”} \\
 & & | & e\ =\ e & \text{comparaison binaire} \\
 & & | & e\ p\ e & \text{op. binaire} \\
 & & | & \texttt{readInt} & \text{op. “}\texttt{readInt}\text{”} \\
 & & | & \texttt{readChar} & \text{op. “}\texttt{readChar}\text{”} \\
 & & | & \texttt{printInt}(e) & \text{op. “}\texttt{printInt}\text{”} \\
 & & | & \texttt{printChar}(e) & \text{op. “}\texttt{printChar}\text{”} \\
 & & | & \{\ \overline{e}\ \texttt{return}\ e\ \} & \text{block} \\
\text{opérateur} & p & ::= & +|-|*|\div|\%|<|>|\wedge &
\end{array}
$$

Notation pour les listes

Notation	Interprétation
$\overline{x}$	séquence $x_0, \ldots, x_n$ pour $n \in \mathbb{N}$
$x, \overline{x}$	séquence $x, x_0, \ldots, x_n$
$\overline{x} \mapsto \overline{\sigma}$	portée $x_0 \mapsto \sigma_0, \ldots, x_n \mapsto \sigma_n$ pour $n \in \mathbb{N}$
ϵ	portée ou séquence vide

Représenter les noms et leur visibilité

symbole de classe	σ_c	$::=$	$(\overline{t} \,\vert\, \Gamma_f \,\vert\, \Gamma_m)_c$
symbole de champ	σ_f	$::=$	$(T)_f$
symbole de méthode	σ_m	$::=$	$(\overline{T} \mid T)_m$
symbole de paramètre	σ_p	$::=$	$(T)_p$

Les symboles de classe ont trois propriétés :

- La liste de toutes les ancêtres notée $\overline{t}$.
- Les champs (Γ_f) et les méthodes (Γ_m) définis dans cette classe ou hérités d'un ancêtre.

Les symboles de champ, de méthode ou de paramètre définissent leurs types (et celui de leurs paramètres pour les méthodes).

portée des classes	Γ_c	$::=$	$\overline{t \mapsto \sigma_c}$
portée des champs	Γ_f	$::=$	$\overline{a \mapsto \sigma_f}$
portée des méthodes	Γ_m	$::=$	$\overline{a \mapsto \sigma_m}$
portée des paramètres	Γ_p	$::=$	$\overline{a \mapsto \sigma_p}$

Les portées sont des dictionnaires qui

- pour un nom donné
- définissent le symbole correspondant.

La portée des classes a une validité globale : dès qu'elle est construite, elle est disponible partout.

Les portées des champs et méthodes sont dépendantes d'une classe et sont attachées à son symbole.

Les portées des paramètres sont locales et n'existent que temporairement, lors de l'analyse d'une méthode.

Le type d'une expression

Les jugements sur le type d'une expression ont tous la forme :

$$\Gamma_c\,;\,\Gamma_p \vdash e : T$$

- e est l'expression concernée par le jugement.
- Γ_c définit toutes les classes du programme.
- Γ_p définit les paramètres de la méthodes pour laquelle l'expression est définie, ou est vide.
- T est le type de l'expression.

On définit des règles d'inférences pour des jugements de cette forme couvrant tous les cas valides.

La première règle d'inférence

$$\Gamma_c\,;\,\Gamma_p \vdash n : \texttt{Int}$$

nous dit que

- quelles que soient les classes du programme,
- quelles que soient les paramètres définis couramment,
- une expression composée d'un nombre quelconque
- a la type `Int`.

D'autres règles simples :

$$\Gamma_c\,;\,\Gamma_p \vdash \texttt{null} : \texttt{Null} \qquad \Gamma_c\,;\,\Gamma_p \vdash \texttt{readInt} : \texttt{Int}$$

$$\Gamma_c\,;\,\Gamma_p \vdash \texttt{readChar} : \texttt{Int}$$

Cette règle avec prémisses

$$\frac{a \mapsto (T)_p \in \Gamma_p}{\Gamma_c\,;\,\Gamma_p \vdash a : T}$$

nous dit que

- quelles que soient les classes du programme,
- une expression composée d'un identifiant,
- pour lequel il existe un symbole dans la portée courante des paramètres
- a le type défini pour ce paramètre dans la portée.

Une autre règle avec prémisses :

$$\frac{t \mapsto (\overline{t}\,|\,\Gamma_f\,|\,\Gamma_m)_c \in \Gamma_c}{\Gamma_c\,;\,\Gamma_p \vdash \texttt{new}\ t : t}$$

De la nature syntaxique des règles

Remarquons une propriété intéressante de ces règles :

Si l'on connaît seulement l'expression dans un jugement sur le type des expressions, on peut choisir une *règle unique* à appliquer.

- Ça ne marche dans aucun autre sens (si on connaît seulement le type, on ne peut pas choisir la règle).
- Le jugement est dit "*syntaxiquement dirigé*".
- Il peut efficacement être implanté comme programme.

Cette règle

$$\frac{\Gamma_c\,;\,\Gamma_p \vdash e : \texttt{Int} \qquad \Gamma_c\,;\,\Gamma_p \vdash e' : \texttt{Int}}{\Gamma_c\,;\,\Gamma_p \vdash e\,p\,e' : \texttt{Int}}$$

nous dit que

- quelles que soient les classes ou les paramètres,
- une opération binaire quelconque entre deux sous-expressions,
- a le type `Int`
- mais seulement si les les deux sous-expressions ont aussi le type `Int` (règle récursive).

Si l'une des sous-expression n'a pas le type `Int`, il n'existe pas de solution au jugement sur le type d'une expression et

- le programme est *invalide du point de vue du typage*,
- car il n'existe pas d'autre règle qui pourrait s'appliquer à l'opération binaire.

D'autres règles avec des prémisses récursives

$$\frac{\Gamma_c\,;\,\Gamma_p \vdash e : \texttt{Int}}{\Gamma_c\,;\,\Gamma_p \vdash \texttt{printInt}(e) : \texttt{Int}} \qquad \frac{\Gamma_c\,;\,\Gamma_p \vdash e : \texttt{Int}}{\Gamma_c\,;\,\Gamma_p \vdash \texttt{printChar}(e) : \texttt{Int}}$$

$$\frac{\Gamma_c\,;\,\Gamma_p \vdash \overline{e} : \overline{U} \qquad \Gamma_c\,;\,\Gamma_p \vdash f : T}{\Gamma_c\,;\,\Gamma_p \vdash \{\ \overline{e}\ \texttt{return}\ f\ \} : T}$$

Notation	Interprétation
$\Gamma_c\,;\,\Gamma_p \vdash \overline{e} : \overline{T}$	$\forall e, T \in \overline{e} \times \overline{T}\,.\,\Gamma_c\,;\,\Gamma_p \vdash e : T$

Le type du jugement sur l'expression-block n'est pas concret, mais est le même que sa dernière expression, quelle qu'elle soit.

$$\frac{\begin{array}{c}\Gamma_c\,;\,\Gamma_p \vdash e : t \\ t \mapsto (\overline{t}\,|\,\Gamma_f\,|\,\Gamma_m)_c \in \Gamma_c \qquad a \mapsto (U)_f \in \Gamma_f\end{array}}{\Gamma_c\,;\,\Gamma_p \vdash e.a : U}$$

Cette règle

$$\frac{\begin{array}{c}\Gamma_c\,;\Gamma_p \vdash e : t \qquad t \mapsto (\overline{t}\,|\,\Gamma_f\,|\,\Gamma_m)_c \in \Gamma_c \\ a \mapsto (\overline{U}\,|\,U)_m \in \Gamma_m \qquad \Gamma_c\,;\Gamma_p \vdash \overline{e} : \overline{T} \qquad \Gamma_c \vdash \overline{T} \lhd \overline{U}\end{array}}{\Gamma_c\,;\Gamma_p \vdash e.a(\overline{e}) : U}$$

contient dans ses prémisses un jugement de forme inconnue.

La *forme d'un jugement* définit quel ensemble de règles le définit.

Les jugements de la forme $\Gamma_c \vdash T \lhd T$ sont définis par les règles de sous-typage.

Notation	Interprétation
$\Gamma_c \vdash \overline{T} \lhd \overline{U}$	$\forall T, U \in \overline{T} \times \overline{U} \,.\, \Gamma_c \vdash T \lhd U$

Cette notation est uniquement définie si $\overline{T}$ et $\overline{U}$ sont de même longueur.

Règles de sous-typage

$$\frac{t \mapsto (\overline{t}\,|\,\Gamma_f\,|\,\Gamma_m)_c \in \Gamma_c \qquad u \in \overline{t} \vee u = t}{\Gamma_c \vdash t \lhd u} \qquad \Gamma_c \vdash \mathtt{Null} \lhd t$$

$$\Gamma_c \vdash \mathtt{Null} \lhd \mathtt{Null} \qquad\qquad \Gamma_c \vdash \mathtt{Int} \lhd \mathtt{Int}$$

Les jugements sur le sous-typage sont syntaxiquement dirigés

- pas sur la syntaxe des expressions,
- mais sur la syntaxe des types t et u.

Revenons à nos expressions :

$$\frac{\begin{array}{c}\Gamma_c\,;\Gamma_p \vdash e : t \qquad t \mapsto (\overline{t}\,|\,\Gamma_f\,|\,\Gamma_m)_c \in \Gamma_c \\ a \mapsto (\overline{U}\,|\,U)_m \in \Gamma_m \qquad \Gamma_c\,;\Gamma_p \vdash \overline{e} : \overline{T} \qquad \Gamma_c \vdash \overline{T} \lhd \overline{U}\end{array}}{\Gamma_c\,;\Gamma_p \vdash e.a(\overline{e}) : U}$$

$$\frac{\begin{array}{c}\Gamma_c\,;\Gamma_p \vdash e : t \qquad t \mapsto (\overline{t}\,|\,\Gamma_f\,|\,\Gamma_m)_c \in \Gamma_c \\ a \mapsto (U)_f \in \Gamma_f \qquad \Gamma_c\,;\Gamma_p \vdash f : T \qquad \Gamma_c \vdash T \lhd U\end{array}}{\Gamma_c\,;\Gamma_p \vdash e.a \;=\; f : t}$$

$$\frac{\begin{array}{c}\Gamma_c\,;\Gamma_p \vdash e : T \\ \Gamma_c\,;\Gamma_p \vdash e' : T' \qquad \Gamma_c \vdash T \lhd T' \vee \Gamma_c \vdash T' \lhd T\end{array}}{\Gamma_c\,;\Gamma_p \vdash e \;=\; e' : \mathtt{Int}}$$

La dernière règle sur les expression :

$$\frac{\begin{array}{c}\Gamma_c\,;\Gamma_p \vdash e : \mathtt{Int} \\ \Gamma_c\,;\Gamma_p \vdash f : U \qquad \Gamma_c\,;\Gamma_p \vdash f' : U' \qquad \Gamma_c \vdash U \lhd T \rhd U'\end{array}}{\Gamma_c\,;\Gamma_p \vdash \mathtt{if}\ e\ \mathtt{then}\ f\ \mathtt{else}\ f' : T}$$

utilise le jugement sur les bornes supérieures qui n'a qu'un règle :

$$\frac{\Gamma_c \vdash T' \lhd U \qquad \forall U' \,.\, \left.\begin{array}{c}\Gamma_c \vdash T \lhd U \\ \Gamma_c \vdash T \lhd U' \\ \Gamma_c \vdash T' \lhd U'\end{array}\right\} \Rightarrow \Gamma_c \vdash U \lhd U'}{\Gamma_c \vdash T \lhd U \rhd T'}$$

La borne supérieure U, appelée *LUB* de deux types

- est l'un, si l'autre en est un sous-type ou
- est un nouveau type qui est à la fois sur-type de l'un et l'autre,
- n'est pas définie seulement si ce dernier n'existe pas.

Ceci permet une grande liberté dans l'utilisation du `if`.

L'ennui avec la borne supérieure

Les règles que nous avons vues avant celle du *LUB*

- peuvent être résolues en résolvant séparément chaque jugement des prémisses
- dans un ordre précis quand un jugement dépend d'un autre.

La règle du *LUB*

- doit deviner la valeur U qu'aucun jugement dans les prémisses ne fournit.
- Les prémisses ne peuvent que dire si le U choisi est correct.

Cette règle est dite "*déclarative*" (par opposition aux autres règles algorithmiques).

Pour l'implanter, il faudra trouver un algorithme correspondant qui n'aura potentiellement rien à voir avec la règle.

Retour sur l'analyse de noms

Le jugement sur le type d'une expression demande que

- des symboles soient définis pour les classes, les champs, les méthodes et les paramètres
- que ces symboles soient introduis dans des portées appropriées pour les faire correspondre à des noms.

Ceci correspond à la tâche de *l'analyse de noms*.

Au lieu de l'approche informelle du cours, nous allons formaliser l'analyse de noms par des jugements sur les programmes, les classes et les membres. Ceci à l'avantage

- de mieux s'intégrer avec le reste du système de type
- de régler un tas de détails de l'implantation qui sont laissés flous dans la version informelle.

Attention toutefois : c'est assez subtile !

L'entier du travail à faire est contenu dans le jugement sur la validité d'un programme, défini par la règle suivante.

$$\frac{\begin{array}{c}\mathtt{none} \mapsto (\epsilon \,|\, \epsilon \,|\, \epsilon)_c \vdash \overline{D} \;\Rightarrow\; \Gamma'_c \\ \Gamma'_c \vdash \overline{D} \;\Rrightarrow\; \Gamma_c \qquad \Gamma_c \vdash \overline{D} \diamond \qquad \Gamma_c \,;\, \epsilon \vdash e : T\end{array}}{\overline{D}\; e \diamond}$$

Chaque jugement des prémisses définit un tâche :

1. La portée des classes est calculée. L'extension de classe est validée. Les symboles de classe $(\overline{t} \,|\, \Gamma_f \,|\, \Gamma_m)_c$ sont incomplets.
2. Les symboles de classes sont complétés.
3. La déclaration des membres est validée par rapport aux autres et à l'héritage. Les expressions des membres sont validées.
4. L'expression principale est aussi validée.

Notez comme les dépendances entre prémisses définissent un ordre d'évaluation de celles-ci.

La prémisse $\mathtt{none} \mapsto (\epsilon \,|\, \epsilon \,|\, \epsilon)_c \vdash \overline{D} \;\Rightarrow\; \Gamma'_c$ est définie par la règle :

$$\frac{s \mapsto (\overline{s} \,|\, \epsilon \,|\, \epsilon)_c \in \Gamma_c \qquad \Gamma'_c = \Gamma_c \uplus t \mapsto (s, \overline{s} \,|\, \epsilon \,|\, \epsilon)_c}{\Gamma_c \vdash \mathtt{class}\; t \lhd s \,\{\, \overline{d} \,\} \;\Rightarrow\; \Gamma'_c}$$

- Une classe ne peut être définie qu'une seule fois.
- Une classe n'hérite que de classes définies précédemment.
- `none` est la super-classe commune.
- Notez la façon dont la liste des ancêtres est construite.

Notation	Interprétation
$\Gamma_c \vdash \overline{D} \;\Rightarrow\; \Gamma'_c$	$\begin{cases} \Gamma_c \vdash D_0 \;\Rightarrow\; \Gamma^0_c \\ \vdots \\ \Gamma^{n-1}_c \vdash D_n \;\Rightarrow\; \Gamma'_c \end{cases}$
$\Gamma \uplus x \mapsto \sigma$	$\Gamma, x \mapsto \sigma$ si $\forall \Gamma', \Gamma'', \sigma' \,.\, \Gamma \neq (\Gamma', x \mapsto \sigma', \Gamma'')$

La prémisse $\Gamma'_c \vdash \overline{D} \Rrightarrow \Gamma_c$ est définie par la règle :

$$\frac{s \mapsto (\overline{s}\,|\,\Gamma_f\,|\,\Gamma_m)_c \in \Gamma_c \qquad \Gamma'_f = \Gamma_f + \mathit{fields}(\overline{d}) \qquad \Gamma'_m = \Gamma_m + \mathit{methods}(\overline{d}) \qquad \Gamma'_c = \Gamma_c + t \mapsto (s, \overline{s}\,|\,\Gamma'_f\,|\,\Gamma'_m)_c}{\Gamma_c \vdash \mathtt{class}\ t \lhd s\,\{\,\overline{d}\,\} \Rrightarrow \Gamma'_c}$$

- La portée des champs contient les champs hérités.
- Si un champ redéfinit un champ hérité, ce dernier est remplacé dans la portée par le nouveau.
- Idem pour les méthodes.

Notation	Interprétation
$\Gamma + x \mapsto \sigma$	$\begin{cases} \Gamma', x \mapsto \sigma, \Gamma'' & \text{si } \exists \Gamma', \Gamma'', \sigma' \,.\, \Gamma = (\Gamma', x \mapsto \sigma', \Gamma'') \\ \Gamma, x \mapsto \sigma & \text{sinon} \end{cases}$

Les autres notations pour la déclaration des membres sont :

Notation	Interprétation
$\Gamma_c \vdash \overline{D} \Rrightarrow \Gamma'_c$	$\begin{cases} \Gamma_c \vdash D_0 \Rrightarrow \Gamma^0_c \\ \vdots \\ \Gamma^{n-1}_c \vdash D_n \Rrightarrow \Gamma'_c \end{cases}$
$\mathit{fields}(\overline{d})$	$\biguplus_{\mathtt{field}\ a:T\,=\,e\,\in\,\overline{d}} a \mapsto (T)_f$
$\mathit{methods}(\overline{d})$	$\biguplus_{\mathtt{method}\ a\,(\overline{a}:\overline{T}):T\,=\,e\,\in\,\overline{d}} a \mapsto (\overline{T}\,|\,T)_m$

- Il ne peut pas y avoir plusieurs champs ou méthodes de même nom dans la même classe.

La prémisse $\Gamma_c \vdash \overline{D} \diamond$ est définie par la règle :

$$\frac{\Gamma_c\,;\,\mathtt{this} \mapsto (t)_p \vdash \overline{d} \diamond}{\Gamma_c \vdash \mathtt{class}\ t \lhd s\,\{\,\overline{d}\,\} \diamond}$$

Qui renvoie le travail au jugement sur la validité d'un membre.

- La portée des paramètres existe dès ce niveau.
- Chaque champ et méthode est validée avec un paramètre `this` du type de la classe défini dans la portée.

Notation	Interprétation
$\Gamma_c \vdash \overline{D} \diamond$	$\forall D \in \overline{D} \,.\, \Gamma_c \vdash D \diamond$
$\Gamma_c\,;\,\Gamma_p \vdash \overline{d} \diamond$	$\forall d \in \overline{d} \,.\, \Gamma_c\,;\,\Gamma_p \vdash d \diamond$

Pour le champ, le jugement sur la validité des membres doit

- vérifier que le type donné au champ soit bien un type valide,
- vérifier que le champ à le même type que le champ qu'il redéfinit (si redéfinition il y a),
- vérifier que l'expression assignée au champ soit d'un type compatible avec le champ.

En plus, pour la méthode, il doit

- vérifier la validité des types données au paramètres,
- ajouter les paramètres à la portée des paramètres,
- vérifier la redéfinition en tenant compte des paramètres.

Soit $T \lhd U \lhd V$. Une classe définit $\texttt{method}\ f\ (\ a : U\ ) : U\ =\ \ldots$

- Une sous-classe définit $\texttt{method}\ f\ (\ a : U\ ) : T\ =\ \ldots$ chaque T étant un U, n'importe quel appel au f original se satisfera d'un T en retour.

Pour le résultat des méthodes redéfinies, T est *co-variant* à U.

- Une sous-classe définit $\texttt{method}\ f\ (\ a : V\ ) : U\ =\ \ldots$ chaque T étant aussi un V, appeler f avec un T sera correct.

Pour les paramètres des méthodes, V est *contra-variant* à U.

Un champ doit être du même type que celui qu'il redéfinit.

- Comme on peut lire *et* écrire dans un champ, on peut le considérer comme simultanément paramètre et résultat.
- Seul T est simultanément co- et contra-variant à T.
- On dit que la redéfinition des champs est invariante.

La règle pour le champ est la suivante :

$$\frac{\begin{array}{c}\texttt{this} \mapsto (t)_p \in \Gamma_p \\ \left.\begin{array}{r} t \mapsto (s, \overline{s} \,|\, \Gamma_f \,|\, \Gamma_m)_c \in \Gamma_c \\ s \mapsto (\overline{s} \,|\, \Gamma'_f \,|\, \Gamma'_m)_c \in \Gamma_c \\ a \mapsto (T')_f \in \Gamma'_f \end{array}\right\} \Rightarrow T = T' \\ \Gamma_c \vdash T \diamond \qquad \Gamma_c\,;\Gamma_p \vdash e : U \qquad \Gamma_c \vdash U \lhd T \end{array}}{\Gamma_c\,;\Gamma_p \vdash \texttt{field}\ a : T\ =\ e \diamond}$$

La règle pour la méthode est la suivante :

$$\frac{\begin{array}{c}\texttt{this} \mapsto (t)_p \in \Gamma_p \\ \left.\begin{array}{r} t \mapsto (s, \overline{s} \,|\, \Gamma_f \,|\, \Gamma_m)_c \in \Gamma_c \\ s \mapsto (\overline{s} \,|\, \Gamma'_f \,|\, \Gamma'_m)_c \in \Gamma_c \\ a \mapsto (T' \,|\, \overline{T'})_m \in \Gamma'_m \end{array}\right\} \Rightarrow \Gamma_c \vdash T \lhd T' \wedge \Gamma_c \vdash \overline{T'} \lhd \overline{T} \\ \Gamma_c \vdash T \diamond \\ \Gamma_c \vdash \overline{T} \diamond \qquad \Gamma_c\,;\Gamma_p + params(\overline{a} : \overline{T}) \vdash e : U \qquad \Gamma_c \vdash U \lhd T \end{array}}{\Gamma_c\,;\Gamma_p \vdash \texttt{method}\ a\ (\ \overline{a} : \overline{T}\ ) : T\ =\ e \diamond}$$

Notation	Interprétation
$\Gamma_c \vdash \overline{T} \diamond$	$\forall T \in \overline{T} \,.\, \Gamma_c \vdash T \diamond$
$\Gamma_c \vdash \overline{T} \lhd \overline{U}$	$\forall T, U \in \overline{T} \times \overline{U} \,.\, \Gamma_c \vdash T \lhd U$
$params(\overline{a} : \overline{T})$	$\underset{a:T \,\in\, \overline{a}:\overline{T}}{+} a \mapsto (T)_p$

Comme les types que l'utilisateur donne aux champs et méthodes ne sont pas calculées, on ne peut pas leur faire confiance :

$$\frac{t \mapsto (\overline{t} \,|\, \Gamma_f \,|\, \Gamma_m)_c \in \Gamma_c}{\Gamma_c \vdash t \diamond} \qquad \Gamma_c \vdash \texttt{Null} \diamond \qquad \Gamma_c \vdash \texttt{Int} \diamond$$

- On vérifie simplement qu'une classe utilisée comme type à bien été définie dans le programme.

Pour conclure

Vous avez de la chance ...

- le système de type de Vier est syntaxiquement dirigé et principalement algorithmique.
- L'implantation consiste à faire correspondre des concepts de programmations aux concepts de notation
- mais pour le reste, la spécification définit tous les détails.

Ce type de spécification est d'une grande valeur en général.

- Beaucoup des problèmes sont naturellement représenté par des jugements.
- Ce genre de formalisme est très utile pour faire des preuves.
- Si les règles sont dirigée par la bonne structure et algorithmiques, l'implantation est facile.