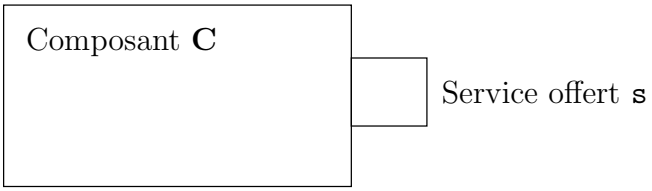
Sémantique formelle du plus simple composant Kmelia

Pascal Sotin

14 mai 2009



1 Description du système

Le composant **C** contient :
- Un ensemble V de variables identifiées par leur nom. Chacune a un domaine, donné par la fonction *dom* qui associe à un nom de variable (dans V) le domaine de la variable.
- Un état. L'état du composant appartient à Σ_V (espace d'état généré par un ensemble de variables typées : voir annexe A.1). On note abusivement cet état, propre au composant, $\Sigma_{\mathbf{C}}$.
- Une séquence d'initialisation, $\mathtt{ini}_{\mathbf{C}}$. L'initialisation ne contient que des actions élémentaires (sans communication) et doit valuer chaque variable.
- Un prédicat invariant $inv_{\mathbf{C}} \subseteq \Sigma_{\mathbf{C}}$. Comme précisé en annexe A.2, on manipule de façon transparente un prédicat par un ensemble. L'application de la substitution[1] associée à cette séquence d'initialisation établit l'invariant : $[\mathtt{ini}_{\mathbf{C}}]\, inv_{\mathbf{C}}$.
- Le service `s`.

Le service `s` comporte :
- Un nom, un ensemble P de paramètres. On note $dom(p)$ le type du paramètre p (de P). On note $result(\mathtt{s})$ le type de la valeur de retour.
- Un ensemble L de variables locales. Typées elles aussi.
- Un LTS. Nous décrivons plus loin ce LTS, mais dont nous mentionnons déjà l'ensemble d'état Q et en particulier l'état initial q_0.
- Par abus de notation, on note $\Sigma_{\mathtt{s}}$ l'espace d'état propre au service. On a $\Sigma_{\mathtt{s}} = \Sigma_P \times \Sigma_L \times Q \times option(result(\mathtt{s}))$.

[1] Voir la théorie des substitutions B.

– Un prédicat de précondition, $pre \subseteq \Sigma_{\mathbf{C}} \times \Sigma_P$.
– Une séquence d'initialisation, $\texttt{ini}_{\texttt{s}}$. Cette séquence ne contient pas de communication et doit valuer chaque variable locale.
– Un prédicat de postconditions, $post \subseteq (\texttt{before} : \Sigma_{\mathbf{C}}) \times \Sigma_P \times \Sigma_{\mathbf{C}} \times result(\texttt{s})$.

Le LTS du service `s` comporte :
– Un ensemble Q d'états d'automate.
– Un état $q_0 \in Q$ dit initial et un ensemble $Q_F \subseteq Q$ d'état finaux.
– Un ensemble G de gardes.
– Un ensemble A d'actions.
– Une relation de transition, $\delta \subseteq Q_{\overline{F}} \times G \times A \times Q$.

Une action, mentionnée sur une transition du LTS peut être :
– Un appel de service. Mais il n'y a pas ici de services requis, donc personne à appeler.
– Un retour de service, noté $\texttt{??s}(v)$ où v est une variable connue, globale, locale ou paramètre. On pourrait supposer que cette transition donne toujours sur un état final et bloquant, dans le cas d'un service non partagé.
– Une alteration de l'état des variables par une fonction f. On acceptera aussi tout langage de programmation dont on peut interpréter le code par une sémantique dénotationnelle. Exemple : `x:=5;`.
– Une lecture ou une écriture sur un canal. Ici, le seul canal existant est celui ouvert par l'appelant, par convention nommé Caller. La lecture est sous la forme Canal?message$(e_1, \ldots, e_n)$ ou les e_i sont des expressions. L'écriture est similaire, mais le `?` devient un `!` et les expressions sont remplacées par des variables.

2 Sémantique

On décrit la sémantique d'un appel du service du composant. Comme il n'y a qu'un seul service, et qu'il n'est pas partagé, il ne peut être lancé simultanément qu'une seule fois par l'appellant.

Un état de la sémantique est de la forme :

$$\{\omega\} \mid \Sigma_{\mathbf{C}} \times option(\Sigma_{\texttt{s}}) \mid \{\tau \langle \text{e} \rangle \mid e \in \text{ErrMsg}\}$$

ou de façon plus détaillée :

$$\{\omega\} \mid \Sigma_V \times option(\Sigma_P \times \Sigma_L \times Q \times option(result(\texttt{s}))) \mid \{\tau \langle \text{e} \rangle \mid e \in \text{ErrMsg}\}$$

L'état initial de notre système est ω.

2.1 Démarrage du composant

$$\text{start} \frac{\begin{array}{c}\text{Démarrage du composant } \mathbf{c} \\ val(v) \in [\![\texttt{ini}_{\mathbf{C}}]\!]_{\text{ini}} \wedge v \in inv_{\mathbf{C}}\end{array}}{\omega \Rightarrow (v, none)}$$

$$\text{start inv fail}\ \frac{\text{Démarrage du composant } \mathbf{c} \quad val(v) \in [\![\mathtt{ini}_{\mathbf{C}}]\!]_{\text{ini}} \wedge v \notin inv_{\mathbf{C}}}{\omega \Rightarrow \tau\,\langle \text{L'initialisation n'établit pas l'invariant}\rangle}$$

$$\text{start code fail}\ \frac{\text{Démarrage du composant } \mathbf{c} \quad err\,\langle \text{Valuation partielle}\rangle \in [\![\mathtt{ini}_{\mathbf{C}}]\!]_{\text{ini}}}{\omega \Rightarrow \tau\,\langle \text{Valuation incomplète}\rangle}$$

$$\text{start uncomplete ini}\ \frac{\text{Démarrage du composant } \mathbf{c} \quad [\![\mathtt{ini}_{\mathbf{C}}]\!]^{\sharp}_{\text{ini}} \top \neq \bot}{\omega \Rightarrow \tau\,\langle \text{Echec du code d'initialisation}\rangle}$$

Au démarrage d'un composant, son code d'initialisation est exécuté. Il doit initialiser explicitement l'ensemble des variables contenues dans V. De plus la valeur obtenue doit satisfaire l'invariant du composant.

La garantie d'avoir initialisé toutes les variables est fournie par une analyse type de flôt de données (voir Annexe B.1).

2.2 Démarrage du service

$$\text{call}\ \frac{\text{Démarrage du service } \mathtt{s}(p) \quad (v,p) \in pre \qquad l \in ini_{\mathtt{s}}}{(v, none) \Rightarrow (v, some\,(p, l, q_0, none))}$$

$$\text{bad call}\ \frac{\text{Démarrage du service } \mathtt{s}(p) \quad (v,p) \notin pre}{(v, none) \Rightarrow \tau\,\langle \text{appel illégal}\rangle}$$

Le démarrage de service est légal, si l'état actuel du composant et les paramètres de l'appel satisfont la précondition. Dans le cas contraire, on obtient une erreur. Ce choix sémantique suppose qu'une vérification dynamique des préconditions est effectuée (quelles possibilités de garanties statiques ?). Le démarrage du service s'il est déjà en cours n'est pas autorisé. La sémantique est bloquante pour l'appelant sur ce point (vérification statique de non-blocage au démarrage d'un services ?).

2.3 Arrêt du service

$$\text{termination}\ \frac{q \in Q_F \qquad \text{post...}}{(v, some\,(p, l, q)) \Rightarrow (v, none)}$$

$$\text{bad termination}\ \frac{q \in Q_F \qquad \neg\text{post...}}{(v, some\,(p, l, q)) \Rightarrow \tau\,\langle \text{Violation de post-conditions}\rangle}$$

Des questions émérgent :

- Rôle de l'instruction de retour `!!` *vs.*rôle des états finaux.
  - Risques de retour multiple.

  - Le retour débloque l'appellant. Il est succeptible de réappeler immédiatement, avant la fin du service.
- Portée des postconditions.
  - Une post-condition peut elle parler de la valeur de retour ? De la valeur des locales ?
- Synchronisation avec l'appelant. Le service termine-t-il si l'appelant n'est pas en attente de retour (`??`).

2.4 Modification d'état par programme

$$\text{code}\ \frac{q \overset{code}{\rightarrow} q' \qquad val\,\langle v', l'\rangle = [\![code]\!]_{\text{elem}}(v, l) \qquad v \in inv_{\mathbf{C}} \qquad (c, p, l, q) \in inv_{\mathbf{s}}}{(v, some\,(p, l, q)) \Rightarrow (v', some\,(p, l', q'))}$$

$$\text{echec code}\ \frac{q \overset{code}{\rightarrow} q' \qquad err\,\langle reason\rangle = [\![code]\!]_{\text{elem}}(v, l)}{(v, some\,(p, l, q)) \Rightarrow \tau\,\langle reason\rangle}$$

$$\text{bad code}\ \frac{q \overset{code}{\rightarrow} q' \qquad val\,\langle v', l'\rangle = [\![code]\!]_{\text{elem}}(v, l) \qquad (v \in inv_{\mathbf{C}} \vee (c, p, l, q) \in inv_{\mathbf{s}})}{(v, some\,(p, l, q)) \Rightarrow \tau\,\langle \text{Violation d'invariant}\rangle}$$

- Les appels de méthode sont par valeur.
  - Les paramètres sont-ils modifiables ?
- L'invariant porte-t-il sur l'état de l'automate ?

A Formalisme pour la sémantique

A.1 Types manipulés

Nous définissons la structure manipulée par la sémantique à l'aide de types, inspirés des types algébriques de données et des types abstraits de données de ML et Haskell. L'idée est de pouvoir faire du *pattern matching* sur l'état et de pouvoir définir ou supposer définies des fonctions de manipulations de ces types. Un type défini un ensemble, et nous manipulons de façon transparente l'un pour l'autre. Le typage d'une variable x par un type t signifie $x \in t$.

- Soit x une variable typée de Kmelia. $dom(x)$ est son type.
- Soit `s` un service. $result(\texttt{s})$ est le type de la valeur de retour.
- Soit t un type et `qual` une chaîne de caractères. Le type `qual` : t est le type t qualifié par `qual`. Il définit l'ensemble $\{\texttt{qual}(x) \mid x \in t\}$.
- Soit X un ensemble de variables typées de Kmelia. Σ_X est l'espace d'état associé à cet ensemble de variables. Nous ne précisons pas l'ensemble définit par Σ_X mais nous disposons d'opérateur pour manipuler ce genre de valeurs.
- Soit $t_1, \ldots, t_n$ une séquence finie de types. $t_1 \times \cdots \times t_n$ est le produit cartésien de ces types. Il définit l'ensemble $\{(v_1, \ldots, v_n) \mid v_i \in t_i\}$.
- Soit $t_1, \ldots, t_n$ une séquence finie de types deux à deux disjoints. $t_1 | \cdots | t_n$ est l'union de ces types. Il définit l'ensemble $t_1 \cup \cdots \cup t_n$.
- Soit t un type. $[t]$ est le type liste d'éléments de t. Nous adoptons la définition récursive classique, et effectuons le *pattern matching* à l'aide de la liste vide [] et de la décomposition en tête et corps $[x|r]$.
- Soit t un type. $option\,(t)$ définit l'ensemble $\{none\} \cup \{some\,(v) \mid v \in t\}$.

A.2 Prédicats et ensembles

Dans la présente sémantique, on manipule assez librement des prédicats pour des ensembles et vice-versa. Cette équivalence est bien connue et nous en rappelons ici les propriétés principales.

Soit S et P un ensemble et un prédicat équivalents. On a :

- $P(x) = x \in S$.
- $S = \{x \mid P(x)\}$.

On aura donc les équivalence suivantes :

Ensembliste	Logique
$A \subseteq B$	$A \Rightarrow B$
$\emptyset$	*faux*
$A \cup B$	$A \vee B$
$A \cap B$	$A \wedge B$

Dans notre sémantique, les ensembles considérés sont des sous-ensembles d'un espace d'état généré par un jeu X de variables Kmelia typées. On note Σ_X cet espace. Dans notre sémantique, les prédicats considérés portent également sur un ensemble de variables libres. Pour un même ensemble de variables X on

retrouve l'équivalence mentionnée plus haut entre des ensembles de tuples et des prédicats portant sur un sous-ensemble de ces variables.

B Analyses

B.1 Analyse d'initialisation

Notre analyse d'initialisation garantie qu'un code d'initialisation `ini` initialise bien un certain ensemble de variables X.

Ce type d'analyse est classique. Le compilateur `javac` assure que toute variable locale utilisée a été au préalable initialisée. L'option `-Wuninitialized` du compilateur `gcc`, combiné à `-Winit-self`, émet un warning pour des raisons similaires. L'option `-Wunused-variable` du même compilateur émet un warning en présence de variables locales inutilisées hors de leur déclaration.

On utilise le treillis $\mathbf{2} = \{\bot, \top\}$. $\top$ signifie peut-être non-initialisé et $\bot$ signifie initialisé pour sûr. La structure $X \to \mathbf{2}$ est également un treillis, qui décrit l'état d'initialisation des variables de l'ensemble X.

En entrée du code `ini`, toutes les variables de X sont potentiellement non-initialisées, les autres variables sont supposées initialisées. L'analyse vise à prouver qu'en sortie, toutes les variables de X seront initialisées.

On utilise la sémantique la sémantique dénotationelle suivante, $(X \to \mathbf{2}) \to (X \to \mathbf{2})$:

$$
\begin{aligned}
&[\![x\texttt{:=}e]\!]^\sharp_{\text{ini}}\sigma = \sigma[x \mapsto [\![e]\!]^\sharp_{\text{ini-expr}}\sigma] \\
&[\![a\texttt{;}b]\!]^\sharp_{\text{ini}}\sigma = [\![b]\!]^\sharp_{\text{ini}}([\![a]\!]^\sharp_{\text{ini}}\sigma) \\
&[\![\texttt{if } c \texttt{ then } a \texttt{ else } b]\!]^\sharp_{\text{ini}}\sigma = [\![a]\!]^\sharp_{\text{ini}}\sigma \sqcup [\![b]\!]^\sharp_{\text{ini}}\sigma \\
&[\![\texttt{while } c \texttt{ do } a]\!]^\sharp_{\text{ini}}\sigma = \sigma \sqcup [\![a\texttt{; while } c \texttt{ do } a]\!]^\sharp_{\text{ini}}\sigma
\end{aligned}
$$

Et pour les expressions, $(X \to \mathbf{2}) \to \mathbf{2}$:

$$
\begin{aligned}
&[\![x]\!]^\sharp_{\text{ini-expr}}\sigma = \sigma(x) && \text{si } x \text{ est une variable et } x \in X \\
&[\![x]\!]^\sharp_{\text{ini-expr}}\sigma = \bot && \text{si } x \text{ est une variable et } x \notin X \\
&[\![c]\!]^\sharp_{\text{ini-expr}}\sigma = \bot && \text{si } c \text{ est une constante} \\
&[\![\texttt{op}_n \ e_1 \dots e_n]\!]^\sharp_{\text{ini-expr}}\sigma = \bigsqcup_{i \in 1..n} [\![e_i]\!]^\sharp_{\text{ini-expr}}\sigma && \text{si } \texttt{op}_n \text{ est un opérateur } n\text{-aire} \\
&[\![f(e_1, \dots, e_n)]\!]^\sharp_{\text{ini-expr}}\sigma = \bigsqcup_{i \in 1..n} [\![e_i]\!]^\sharp_{\text{ini-expr}}\sigma && \text{si } f \text{ est une fonction totale}
\end{aligned}
$$

La propriété que l'on souhaite montrer est $[\![code]\!]^\sharp_{\text{ini}}\top = \bot$.

Pour cela, on va éliminer les problèmes de point fixe dans notre sémantique dénotationelle grace à la proposition (que l'on ne prouve pas ici) :

$$\forall \sigma \in (X \to \mathbf{2}) \quad [\![\texttt{while } c \texttt{ do } a]\!]^\sharp_{\text{ini}}\sigma = \sigma \sqcup [\![a]\!]^\sharp_{\text{ini}}\sigma$$