Programmation fonctionnelle et système de types – projet

Nathanaël Courant

16 février 2018

1 Résumé

Les fonctionnalités suivantes ont été ajoutées au compilateur :

— Défonctionalisation légère : lors de l'étape de défonctionalisation, une fonction qui n'a pas de variables libres n'est pas considérée comme une variable locale d'une fonction dans laquelle elle est contenue, et la clôture de cette dernière fonction ne contient pas la première. Ceci est réalisé dans `Defun.ml`.

— Application efficace d'une fonctions à plusieurs arguments : une application de fonction à n arguments à n valeurs ne construit pas de clôture intermédiare. Pour cela, le langage intermédiaire `Tail` a été étendu avec la construction supplémentaire `ContCall`, dont le comportement dépend du nombre d'arguments fourni par rapport à l'arité attendue de la fonction à appliquer. Sa sémantique est documentée dans `Tail.ml`, et son introduction a conduit à la modification de `CPS.ml` et `Defun.ml`.

— Types de données algébriques et pattern matching : le langage est étendu en fournissant des tuples, des types de données algébriques, et du pattern matching, y compris profond. Ce changement est a nécessité une modification de la plupart des fichiers, mais le plus intéressant est sans contestation dans `CPS.ml`, qui compile le pattern matching en transformation CPS.

— Effets algébriques : ceux-ci permettent en particulier de créer des opérateurs de contrôle délimité, ou des exceptions. Aucun des deux n'est fourni par défaut cependant, leur implantation est à la charge de l'utilisat·eur·rice (de manière générale, rien n'est fourni par défaut : il faut soi-même définir des listes si on souhaite les utiliser). À nouveau, une grande partie de la chaîne de compilation est modifiée, mais la partie intéressante est dans `CPS.ml` qui se charge d'éliminer complètement les effets lors de la traduction en CPS. Cette dernière est très largement basée sur [2].

— Garbage collector : le programme compilé est équipé d'un garbage collector. Il repose, comme celui d'OCaml, sur des entiers de seulement 63 bits permettant ainsi de distinguer les entiers des pointeurs. Il est prévu pour être générationnel, mais seul la gestion du tas mineur est implantée. Cependant, même avec un tas mineur extrêmement petit (2048 mots de 8 octets), les essais pratiques montre que celui-ci se révèle très efficace, permettant de récupérer près de 95% des mots alloués. Il se trouve ma-

joritairement dans `prologue.h`, avec `Finish.ml` gérant l'interface entre code compilé et le garbage collector.

— Inférence de type statique : le programme est typé statiquement, avec inférence de type et des effets algébriques. Ainsi, un programme lançant des effets non rattrapés n'est pas accepté. La lourdeur caractéristique des déclarations de fonctions de Java est évitée grâce à l'inférence de ces effets, ainsi que des types en général. Comme la présence d'effets nous conduit à du sous-typage, tout le typage est effectué en utilisant *ML-sub*, présenté dans [1]. Les règles de typage (mais pas d'inférence) sont également présentées ci-dessous. On pourra se référer à la présentation de *MLsub* pour plus de détails sur l'algorithme d'inférence de types. Le code implantant ces fonctionnalités est présent dans `Cook.ml` et `Lambda.ml`. Une limite à l'implantation actuelle est qu'elle ne fait pas de vérification d'exhaustivité mais se contente d'un test d'irréfutabilité ; en particulier, un programme qui rattrape différentes instances d'un effet en plusieurs cas et pas un seul ne sera pas marqué comme récupérant l'effet en question. Cependant, cela n'affecte pas la sûreté du typage, mais cause uniquement l'inférence d'un type moins général que possible. On peut étudier les types inférés par le typeur en regardant le fichier *test*`.err`, pour un fichier d'entrée *test*`.lambda`.

Note : Suite à l'introduction des nouvelles fonctionnalités, les fichiers de test déjà fournis ont été modifiés : ainsi, `bool.lambda` a été augmenté de l'annotation `#pragma disable_type_checking` qui désactive le typage (puisque celui-ci n'est pas typable dans le système de types utilisé) ; dans les autres fichiers, les `let ... in` au niveau supérieur ont été remplacés par des `let` dans le but de laisser le typeur afficher les types inférés pour ces fonctions.

2 Système de types

Les types définissables par l'utilisateur sont de la forme :

$$\tau ::= \text{'}\alpha \mid \tau \xrightarrow{\mathcal{E}} \tau \mid \tau * \cdots * \tau \mid ((\tau \mid \mathcal{E}), \ldots, (\tau \mid \mathcal{E}))\ \texttt{constr}$$
$$\mathcal{E} ::= (\texttt{!}\beta \mid []) \sqcup \mathtt{E}_1 \sqcup \cdots \sqcup \mathtt{E}_n$$

Ces types sont ensuite étendus par des types récursifs, des types unions et intersections, ainsi que les types $\top$ et $\bot$, avec des règles de sous-typage structurelles (la seule information de sous-typage intéressante est au niveau des feuilles, pour les effets ; se référer à [1] pour plus de détails sur le sous-typage). Pour les types d'effets, ils sont étendus en utilisant une variable de présence par effet différent pouvant être présent, plus une autre variable de présence comme valeur par défaut pour tous les autres effets. De plus, les types polymorphes sont légèrement modifiés, puisque toute instance d'un type polymorphe peut à présent utiliser un type différent pour les occurrences positives et négatives de chaque variable de type. On pourrait simplifier les types produits en utilisant une seule occurrence dans le cas d'un type covariant ou contravariant, ou même aucune dans le cas d'une variable de type inutile. Cependant, l'implantation actuelle ne fait pas cela (tout le code est présent, sauf l'inférence de si une variable de type est co- ou contravariante).

Cette extension des types permet d'avoir un algorithme d'inférence de types complet, décidable, produisant des types compacts (sans contraintes !) ; tandis

que la restriction des types définissables par l'utilisateur permet d'assurer la polarité des types pour que cet algorithme fonctionne (voir [1]).

On note que les constructeurs de types polymorphes peuvent prendre des arguments qui sont soit des types, soit des effets. Il convient donc de vérifier que les types sont bien sortés. De plus, on remarque qu'il n'y a pas de polymorphisme au niveau des effets.

On définit également :

$$\begin{aligned}
[\Delta_1]\tau_1!\mathcal{E}_1 \leqslant [\Delta_2]\tau_2!\mathcal{E}_2 \;\equiv\;& \tau_1 \leqslant \tau_2 \\
&\wedge \mathcal{E}_1 \leqslant \mathcal{E}_2 \\
&\wedge \text{dom } \Delta_1 \subseteq \text{dom } \Delta_2 \\
&\wedge (\forall x \in \text{dom } \Delta_1, \Delta_2(x) \leqslant \Delta_1(x)) \\
[\Delta_1]\tau_1!\mathcal{E}_1 \leqslant^{\forall} [\Delta_2]\tau_2!\mathcal{E}_2 \;\equiv\;& \exists \rho / \rho([\Delta_1]\tau_1!\mathcal{E}_1) \leqslant [\Delta_2]\tau_2!\mathcal{E}_2
\end{aligned}$$

3 Règles de typage

Par simplicité, on suppose dans la suite que tous les constructeurs de types algébriques et d'effets sont d'arité exactement 1, quitte à remplacer le paramètre par un tuple (éventuellement vide). On suppose également que les pattern matching sont tous complets et ne sont pas profonds, par simplicité. Enfin, le type des constructeurs polymorphes est implicitement instancié (ce qui revient à ajouter une contrainte de sous-typage). Les règles sont explicitées en figure 1.

Références

[1] Stephen Dolan. *Algebraic Subtyping.* Chartered Institute for IT, 2017.

[2] Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. Continuation passing style for effect handlers. 2017.

FIGURE 1 – Règles de typage

$$\textsc{Var-}\Pi\ \frac{\Pi(\hat{\mathrm{x}}) = [\Delta]\tau}{\Pi \Vdash \hat{\mathrm{x}} : [\Delta]\tau!\{\mathtt{E}_l : \bot\}_l} \qquad \textsc{Var-}\Delta\ \frac{}{\Pi \Vdash \mathrm{x} : [\mathrm{x} : \alpha]\alpha!\{\mathtt{E}_l : \bot\}_l}$$

$$\textsc{Abs}\ \frac{\Pi \Vdash e : [\Delta, \mathrm{x} : \tau]\tau'!\mathcal{E}}{\Pi \Vdash \lambda \mathrm{x}.e : [\Delta]\tau \xrightarrow{\mathcal{E}} \tau'!\{\mathtt{E}_l : \bot\}_l} \qquad \textsc{App}\ \frac{\Pi \Vdash e_1 : [\Delta]\tau \xrightarrow{\mathcal{E}} \tau'!\mathcal{E} \qquad \Pi \Vdash e_2 : [\Delta]\tau!\mathcal{E}}{\Pi \Vdash e_1 e_2 : [\Delta]\tau'!\mathcal{E}}$$

$$\textsc{Let}\ \frac{\Pi \Vdash e_1 : [\Delta_1]\tau_1!\mathcal{E} \qquad \Pi, \hat{\mathbf{x}} : [\Delta_1]\tau_1 \Vdash e_2 : [\Delta_2]\tau_2!\mathcal{E}}{\Pi \Vdash \text{let } \hat{\mathbf{x}} = e_1 \text{ in } e_2 : [\Delta_1 \sqcap \Delta_2]\tau_2!\mathcal{E}} \qquad \textsc{Nat}\ \frac{}{\Pi \Vdash n : []\text{int}!\{\mathtt{E}_l : \bot\}_l}$$

$$\textsc{IfZero}\ \frac{\Pi \Vdash e_1 : [\Delta]\text{int}!\mathcal{E} \qquad \Pi \Vdash e_2 : [\Delta]\tau!\mathcal{E} \qquad \Pi \Vdash e_3 : [\Delta]\tau!\mathcal{E}}{\Pi \Vdash \text{ifzero } e_1 \text{ then } e_2 \text{ else } e_3 : [\Delta]\tau!\mathcal{E}}$$

$$\textsc{Tuple}\ \frac{\forall i \in [\![1, n]\!], \Pi \Vdash e_i : [\Delta]\tau_i!\mathcal{E}}{\Pi \Vdash (e_1, \ldots, e_n) : [\Delta](\tau_1 * \cdots * \tau_n)!\mathcal{E}}$$

$$\textsc{MatchTuple}\ \frac{\Pi \Vdash e : [\Delta](\tau_1 * \cdots * \tau_n)!\mathcal{E} \qquad \Pi \Vdash e' : [\Delta, x_1 : \tau_1, \ldots, x_n : \tau_n]\tau!\mathcal{E}}{\Pi \Vdash \text{match } e \text{ with } (x_1, \ldots, x_n) \to e' : [\Delta]\tau!\mathcal{E}}$$

$$\textsc{Constr}\ \frac{\Pi \Vdash e : [\Delta]\tau'!\mathcal{E} \qquad \vdash \mathtt{C} : \tau' \to \tau}{\Pi \Vdash \mathtt{C}\ e : [\Delta]\tau!\mathcal{E}}$$

$$\textsc{MatchConstr}\ \frac{\begin{array}{c}\Pi \Vdash e : [\Delta]\tau'!\mathcal{E} \\ \forall i \in [\![1, n]\!], \vdash \mathtt{C}_i : \tau_i \to \tau' \qquad \forall i \in [\![1, n]\!], \Pi \Vdash e_i : [\Delta, x_i : \tau_i]\tau!\mathcal{E}\end{array}}{\Pi \Vdash \text{match } e \text{ with } \mathtt{C}_1\ x_1 \to e_1 \mid \ldots \mid \mathtt{C}_n\ x_n \to e_n : [\Delta]\tau!\mathcal{E}}$$

$$\textsc{Effect}\ \frac{\Pi \Vdash e : [\Delta]\tau'!\mathcal{E} \qquad \vdash \mathtt{E}_l : \tau' \to \tau}{\Pi \Vdash \mathtt{E}_l\ e : [\Delta]\tau!\{\mathtt{E}_l : \top, (\mathtt{E}_{l'} : \mathcal{E}_{l'})_{l' \neq l}\}}$$

$$\textsc{MatchEffect}\ \frac{\begin{array}{c}\Pi \Vdash e : [\Delta]\tau'!\{(\mathtt{E}_{l_i} : \top)_{i \in [\![1,n]\!]}, (\mathtt{E}_l : \mathcal{E}_l)_{l \notin \{l_1, \ldots, l_n\}}\} \qquad \Pi \Vdash e' : [\Delta, x : \tau']\tau!\mathcal{E} \\ \forall i \in [\![1, n]\!], \vdash \mathtt{E}_{l_i} : \tau_i \to \tau_i' \qquad \forall i \in [\![1, n]\!], \Pi \Vdash e_i : [\Delta, x_i : \tau_i, k_i : \tau_i' \xrightarrow{\mathcal{E}} \tau]\tau!\mathcal{E}\end{array}}{\Pi \Vdash \text{match } e \text{ with } x \to e' \mid \text{effect } \mathtt{E}_{l_1}\ x_1\ k_1 \to e_1 \mid \ldots \mid \text{effect } \mathtt{E}_{l_n}\ x_n\ k_n \to e_n : [\Delta]\tau!\mathcal{E}}$$

$$\textsc{Sub}\ \frac{\Pi \Vdash e : [\Delta]\tau!\mathcal{E} \qquad [\Delta]\tau!\mathcal{E} \leqslant^{\forall} [\Delta']\tau'!\mathcal{E}'}{\Pi \Vdash e : [\Delta']\tau'!\mathcal{E}'}$$