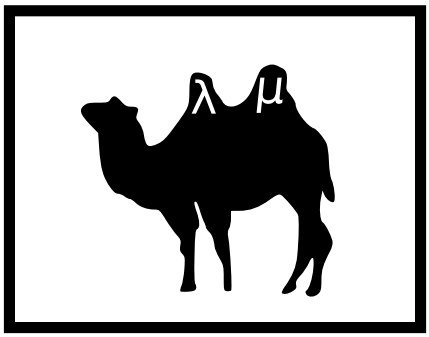
Étude de ctML

ML en appel par nom avec opérateurs de contrôle

Pierre-Marie Pédrot

Stage de recherche de L3 : juin – juillet 2009
École Normale Supérieure de Lyon



Maître de stage : Olivier Laurent (LIP)
Stage réalisé au LIP, dans l'équipe Plume

Remerciements

Je remercie chaleureusement Olivier Laurent, pour m'avoir fourni un sujet de stage fort intéressant et prodigué conseils utiles, attention et littérature *ad-hoc*, et pour m'avoir convié aux différents séminaires de Plume.

Je tiens à remercier Daniel Hirschkoff pour l'intérêt qu'il a manifesté pour CTML, pour son enseignement pédagogique de Prog 1, ainsi que pour avoir nourri à plusieurs reprises les masses désœuvrées du foyer avec les restes des ANR.

Je remercie aussi Matthieu Perrinel, mon compagnon d'infortune, qui est resté de longues heures à mes côtés dans le couloir de la mort surchauffé qui donne sur la machine à café du troisième étage.

Je remercie pêle-mêle : le foyer qui m'a accordé fraîcheur et alimentation, les admissibles de BCPST qui n'ont pas fui mes évocations du $\lambda\mu$-calcul, les spams pléthoriques des stagiaires de la L3IF, et Yves Robert pour ses considérations métaphysiques auprès de la machine à café.

Table des matières

Introduction

Conçu par des mathématiciens à une époque où la distinction entre informatique fondamentale et mathématiques n'était même pas envisageable, le λ-calcul est un reliquat moderne de cette vision duale du monde. Dans le monde de l'informatique, il constitue un modèle éprouvé de calculabilité fondé sur la notion de réécriture. Dans l'univers mathématique, il entretient des relations profondes avec la logique, en ce sens qu'il peut être perçu comme un formalisme pour la preuve. En effet, le λ-calcul sous-tend une logique constructive qualifiée d'*intuitionniste*. Cette dualité est plus connue sous le nom de *correspondance de Curry-Howard*.

Quoique la logique intuitionniste soit incluse dans la logique classique, les logiciens sont longtemps restés perplexes quant à une éventuelle extension classique du λ-calcul. La solution a été observée fortuitement par des informaticiens, qui à la fin des années 1980 se rendirent compte que l'ajout d'opérateurs de contrôle[1] à des langages de programmation dérivés du λ-calcul permettaient d'obtenir des programmes représentant des formules de la logique classique. Cette constatation mena à l'invention du λ_c-calcul de Felleisen dont l'expressivité classique est apportée par l'opérateur $\mathcal{C}$, inspiré par le `call/cc` de Scheme, au prix d'une perte de symétrie des règles logiques.

Il fallut attendre 1992, et l'introduction du $\lambda\mu$-calcul par Parigot [6] pour formaliser une notion de contrôle en λ-calcul qui fasse bon ménage avec la déduction naturelle. Pour ce faire, le $\lambda\mu$-calcul possède deux opérateurs supplémentaires qui correspondent d'un point de vue logique à un choix explicite des séquents, et d'un point de vue fonctionnel à des opérations de sauvegarde et de restauration de la pile du programme.

Comme tout modèle théorique[2], et à l'instar du λ-calcul son ancêtre, le $\lambda\mu$-calcul n'est pas un langage facilement manipulable par un individu lambda. Il a donc fallu inventer des langages plus humains en découlant.

Plusieurs familles de langages de programmation héritant du λ-calcul existent, chacune possédant ses particularités propres. Un des critères de distinction concerne le typage : certains langages fonctionnels sont typés (ML, Haskell), d'autres non (Lisp, Scheme). On distingue aussi les langages en appel par valeur, qui forcent le calcul des argument, de ceux en appel par nom, parfois qualifiés de *paresseux*.

La famille des ML est l'une des plus florissantes, avec deux ressortissants célèbres, d'une part SML et d'autre part Objective Caml. Les ML sont bien connus pour leur typage fort, statique et automatiquement inféré.

Le but de ce stage est de formaliser puis d'étudier une traduction du $\lambda\mu$-calcul dans un dialecte ML, nommé par la suite CTML, comme `catch-throw`-ML, `catch` et `throw` étant les deux opérateurs de contrôle du langage. Une des caractéristiques de ce langage tient au fait que l'on a cherché à conserver le maximum possible de structures provenant de ML, et plus particulièrement d'OCaml.

Une partie du stage ayant consisté à implémenter CTML, on dispose ainsi d'un interpréteur pour notre langage, et même un peu plus. Ce dernier implémente en effet des extensions de CTML qui sont plus utile au programmeur qu'au logicien. Pour faire la différence quand le besoin s'en fera sentir, on nommera CTML la version idéalisée du langage et CTML$^+$ son implémentation avec sucre syntaxique. On trouvera le code de l'interpréteur ainsi que des exemples à l'adresse `http://perso.ens-lyon.fr/pierremarie.pedrot/code/ctml.tar.gz`.

On présentera dans ce rapport les spécifications du langage, aussi bien statiques (structure et typage) que dynamiques (réduction) puis on s'attachera à montrer que CTML vérifie les propriétés qu'on attend d'un ML, à savoir cohérence et correction du typage. On prendra aussi garde à souligner les particularités introduites par nos extensions classiques.

[1] On désigne ainsi tout ce qui permet de manipuler explicitement le flot du calcul : exceptions, primitives de manipulation des continuations, etc.

[2] Peu de personnes peuvent se targuer de programmer sur des machines de Turing...

1 Spécifications du langage

Le langage CTML hérite du ML une certaine quantité de propriétés, en particulier au niveau de la syntaxe et du typage. C'est un langage fonctionnel fortement typé, avec polymorphisme restreint et inférence de type, types somme et filtrage par motifs.

Cependant, contrairement à la plupart des ML, CTML est un langage en appel par nom, et non par valeur. Cela peut se révéler problématique si l'on souhaite rajouter des constructions impératives au langage, puisque l'on a en l'état peu de moyens de prédire et de forcer le calcul[3].

Par ailleurs, et c'est là l'intérêt du langage, CTML possède deux opérateurs de contrôle de premier ordre `catch` et `throw` qui diffèrent sensiblement des systèmes d'exceptions généralement rencontrés dans les langages ML. En outre, CTML implémente un système d'exceptions dont la sémantique est remarquablement différente de celui d'OCaml, entre autres.

1.1 Grammaire

1.1.1 Syntaxe BNF

Étant donnés un ensemble de λ-variables (notées $x, y, \ldots$), un ensemble de μ-variables (notées $\alpha, \beta, \ldots$) et un ensemble de constructeurs (notés C_i^ι, K_i^ι), on définit les termes t de CTML par induction. On considère de même les valeurs v, sous-ensemble des termes de CTML.

$$\begin{array}{rcl} t & ::= & x \mid C_i^\iota \mid K_i^\iota t \mid (t_1, \ldots, t_n) \mid \texttt{fun}\ x \mapsto t \mid t_1 t_2 \mid \texttt{let}\ x = t\ \texttt{in}\ u \mid \texttt{let rec}\ x = t\ \texttt{in}\ u \\ & \mid & \Pi^n\ t\ f \mid \Lambda^\iota\ t\ f_1 \ldots f_k \mid \texttt{catch}\ \pi\ \texttt{in}\ t \mid \texttt{throw}\ \pi\ \texttt{in}\ t \mid (t : \tilde{\sigma}) \\ v & ::= & C_i^\iota \mid K_i^\iota t \mid (t_1, \ldots, t_n) \mid \texttt{fun}\ x \mapsto t \end{array}$$

Tab. 1 – Structure des termes de CTML

On définit de manière analogue la grammaire des types σ et des schémas de types $\tilde{\sigma}$, étant donné un ensemble de variables de type α et un ensemble de types somme ι.

$$\begin{array}{rcl} \sigma & ::= & \alpha \mid \sigma_1 \to \sigma_2 \mid (\sigma_1 * \ldots * \sigma_n) \mid \iota\{\bar{\sigma}\} \\ \tilde{\sigma} & ::= & \forall \bar{\alpha}.\sigma \end{array}$$

Tab. 2 – Structure des types de CTML

※ Afin de faciliter l'écriture des programmes, CTML$^+$ utilise des identifiants ASCII aussi bien pour les λ-variables que pour les μ-variables. La nature d'un identifiant est ainsi déterminée par son emploi, et l'interpréteur retourne une erreur s'il y a confusion dans l'usage. De même, les variables généralisées des schémas de types sont écrites `'a`, `'b`, ... comme en Objective Caml.

1.1.2 Types somme

On se donne un ensemble $\mathcal{T}$ de types de base définis par des constructeurs, avec pour tout $\iota \in \mathcal{T}$, $\iota\{\bar{\alpha}\} = E_1^\iota \mid \ldots \mid E_{k(\iota)}^\iota$ où $E_i^\iota = C_i^\iota$ ou $E_i^\iota = K_i^\iota\{\sigma_i\}$ avec $\mathrm{fv}(\sigma_i) \subseteq \bar{\alpha}$. On suppose en outre que les constructeurs sont distincts deux à deux. Chaque type somme donne lieu à un déconstructeur Λ^ι. Dans la suite, on notera $\iota\{\bar{\alpha}\} = E_1 \mid \ldots \mid E_k$.

Par ailleurs, cette définition n'exclut pas les types mutuellement récursifs ; leur éventuelle prise en compte ne change pas grand chose à la théorie.

[3] Cf. la notation `do` utilisée en Haskell, langage paresseux par excellence, qui use et abuse de monades.

On considèrera en outre quelques types somme habituels qui ont un sucre syntaxique qui leur est propre :

$$\begin{array}{lcl} \texttt{void}\{\emptyset\} & \equiv & \\ \texttt{unit}\{\emptyset\} & \equiv & \texttt{()} \\ \texttt{bool}\{\emptyset\} & \equiv & \texttt{true} \mid \texttt{false} \end{array}$$

※ CTML$^+$ permet de définir ses propres types somme grâce à une syntaxe similaire à celle d'OCaml. Il autorise aussi la déclaration d'abréviations de types.

1.1.3 Filtrage par motif

On conviendra que le filtrage par motifs ne permet que de déconstruire un unique niveau de constructeur, et constitue un sucre syntaxique pour dénoter les déconstructeurs (δ-règles). On a les définitions suivantes :

$$\begin{array}{lcl} \texttt{match}\ t\ \texttt{with}\ (x_1, \ldots, x_n) \mapsto u & \equiv & \Pi^n\ t\ (\texttt{fun}\ x_1 \mapsto \ldots x_n \mapsto u) \\ \texttt{match}\ t\ \texttt{with}\ [E_i \mapsto u_i] & \equiv & \Lambda^\iota\ t\ f_1 \ldots f_k \\ t; u & \equiv & \Lambda^{\texttt{unit}}\ t\ u \\ \texttt{if}\ t\ \texttt{then}\ u_1\ \texttt{else}\ u_2 & \equiv & \Lambda^{\texttt{bool}}\ t\ u_1 u_2 \end{array}$$

On prend dans le cas du type somme $f_i = u_i$ si $E_i = C_i$ et $f_i = \texttt{fun}\ x \mapsto u_i$ si $E_i = K_i x$.

※ De fait la syntaxe de CTML$^+$ est un plus large que cet édulcorant syntaxique ; elle autorise le motif universel et quelques autres subtilités[4].

1.1.4 Exceptions

On définit un sucre syntaxique pour les exceptions en rajoutant les commandes suivantes :

$$\begin{array}{lcll} \texttt{try}\ t\ \texttt{with}\ C \mapsto u & \equiv & \texttt{catch}\ \rho\ \texttt{in}\ t\langle\texttt{raise}\ C \leftarrow \texttt{throw}\ \rho\ \texttt{in}\ u\rangle & \rho \notin \mathrm{fv}(t) \cup \mathrm{fv}(u) \\ \texttt{try}\ t\ \texttt{with}\ Kx \mapsto u & \equiv & \texttt{catch}\ \rho\ \texttt{in}\ t\langle\texttt{raise}\ Kw \leftarrow \texttt{throw}\ \rho\ \texttt{in}\ (\texttt{fun}\ x \mapsto u)w\rangle & \rho \notin \mathrm{fv}(t) \cup \mathrm{fv}(u) \end{array}$$

※ Encore une fois, l'interpréteur est plus large et autorise des déclarations d'exceptions multiples ou universelles[5]. On peut ainsi définir en une seule structure `try-with` plusieurs exceptions et utiliser le motif universel pour déclarer et rattraper n'importe quelle exception dans le code qui suit.

Formellement, CTML$^+$ se comporte comme si on avait :

$$\begin{array}{lcll} \texttt{try}\ t\ \texttt{with}\ p_1 \mid \ldots \mid p_n & \equiv & \texttt{try}\ (\texttt{try}\ t\ \texttt{with}\ p_1)\ \texttt{with}\ p_2 \mid \ldots \mid p_n & \\ \texttt{try}\ t\ \texttt{with}\ \_ \mapsto u & \equiv & \texttt{catch}\ \rho\ \texttt{in}\ t\langle\texttt{raise}\ \_ \leftarrow \texttt{throw}\ \rho\ \texttt{in}\ u\rangle & \rho \notin \mathrm{fv}(t) \cup \mathrm{fv}(u) \end{array}$$

1.2 Typage

Le typage se définit comme une relation $\Gamma \mid \Delta \vdash t : \sigma$ où Γ est un ensemble associatif des λ-variables dans les schémas de types, Δ est un ensemble associatif des μ-variables dans les types, t est un terme et σ est un type.

Pour typer les termes de CTML dans un système à la ML, il a fallu, à l'instar du typage des termes du $\lambda\mu$-calcul, rajouter un environnement de typage des μ-variables. Il semblerait que le polymorphisme à la ML ne cohabite pas bien avec le typage des μ-variables[6] et par conséquent les μ-variables sont monomorphes.

[4] L'interpréteur se comporte comme OCaml vis-à-vis du `;` et du `if then else` .

[5] On pourra par exemple écrire dans un style plus naturel $\texttt{try}\ t\ \texttt{with}\ C_1 \mapsto u_1 \mid K_2 x \mapsto u_2 \mid \_ \mapsto u_3$.

[6] Manifestement parce qu'un type $\sigma \in \Delta$ correspond à $\neg\sigma \in \Gamma$.

1.2.1 Typage du noyau

On fournit ci-dessous l'ensemble des règles de typage pour CTML. Il s'agit d'une extension du système de typage de ML[7] avec des règles orthogonales pour les μ-variables. L'algorithme d'inférence de ML est directement adaptable à ces règles.

On adoptera les notations suivantes :

- $\forall\bar{\alpha}.\sigma \succ \tau$ s'il existe une substitution θ telle que $\mathrm{dom}(\theta) = \bar{\alpha}$ et $\theta(\sigma) = \tau$.
- $\tilde{\sigma} \leq \tilde{\tau}$ si pour tout υ, si $\tilde{\sigma} \succ \upsilon$ alors $\tilde{\tau} \succ \upsilon$.
- $\mathrm{Gen}(\sigma, \Gamma, \Delta) = \forall\bar{\alpha}.\sigma$ où $\bar{\alpha} = \mathrm{fv}(\sigma) \setminus (\mathrm{fv}(\Gamma) \cup \mathrm{fv}(\Delta))$.

$$\frac{(x : \tilde{\sigma}) \in \Gamma \qquad \tilde{\sigma} \succ \sigma}{\Gamma \mid \Delta \vdash x : \sigma}\ (\textsc{Var})$$

$$(\textsc{Sum-}C)\ \frac{C\{\emptyset\} \in \iota\{\bar{\alpha}\}}{\Gamma \mid \Delta \vdash C : \iota\{\bar{\sigma}\}} \qquad \frac{K\{\sigma\} \in \iota\{\bar{\alpha}\} \qquad \Gamma \mid \Delta \vdash t : \sigma\langle\bar{\alpha} \leftarrow \bar{\tau}\rangle}{\Gamma \mid \Delta \vdash Kt : \iota\{\bar{\tau}\}}\ (\textsc{Sum-}K)$$

$$\frac{(\Gamma \mid \Delta \vdash t_i : \sigma_i)_{i \leq n}}{\Gamma \mid \Delta \vdash (t_1, \ldots, t_n) : (\sigma_1 * \ldots * \sigma_n)}\ (\textsc{Tup-}n)$$

$$(\textsc{Fun})\ \frac{\Gamma \sqcup (x : \sigma) \mid \Delta \vdash t : \tau}{\Gamma \mid \Delta \vdash \mathtt{fun}\ x \mapsto t : \sigma \to \tau} \qquad \frac{\Gamma \mid \Delta \vdash t : \sigma \to \tau \qquad \Gamma \mid \Delta \vdash u : \sigma}{\Gamma \mid \Delta \vdash tu : \tau}\ (\textsc{App})$$

$$\frac{\Gamma \mid \Delta \vdash t : \sigma \qquad \Gamma \sqcup (x : \mathrm{Gen}(\sigma, \Gamma, \Delta)) \mid \Delta \vdash u : \tau}{\Gamma \mid \Delta \vdash \mathtt{let}\ x = t\ \mathtt{in}\ u : \tau}\ (\textsc{Let})$$

$$\frac{\Gamma \sqcup (x : \sigma) \mid \Delta \vdash t : \sigma \qquad \Gamma \sqcup (x : \mathrm{Gen}(\sigma, \Gamma, \Delta)) \mid \Delta \vdash u : \tau}{\Gamma \mid \Delta \vdash \mathtt{let\ rec}\ x = t\ \mathtt{in}\ u : \tau}\ (\textsc{Let-Rec})$$

$$\frac{\Gamma \mid \Delta \vdash t : (\sigma_1 * \ldots * \sigma_n) \qquad \Gamma \mid \Delta \vdash f : \sigma_1 \to \ldots \sigma_n \to \tau}{\Gamma \mid \Delta \vdash \Pi^n\ t\ f : \tau}\ (\textsc{Match-Tup-}n)$$

$$\frac{\Gamma \mid \Delta \vdash t : \iota\{\bar{\upsilon}\} \qquad \left(\begin{cases} \Gamma \mid \Delta \vdash f_i : \tau & \text{si } E_i = C_i\{\emptyset\} \\ \Gamma \mid \Delta \vdash f_i : \sigma\langle\bar{\alpha} \leftarrow \bar{\upsilon}\rangle \to \tau & \text{si } E_i = K_i\{\sigma\} \end{cases}\right)_{i \leq k}}{\Gamma \mid \Delta \vdash \Lambda^{\iota}\ t\ f_1 \ldots f_k : \tau}\ (\textsc{Match-Sum-}\iota)$$

$$(\textsc{Catch})\ \frac{\Gamma \mid \Delta \sqcup (\pi : \sigma) \vdash t : \sigma}{\Gamma \mid \Delta \vdash \mathtt{catch}\ \pi\ \mathtt{in}\ t : \sigma} \qquad \frac{\Gamma \mid \Delta \vdash t : \tau \qquad (\pi : \tau) \in \Delta}{\Gamma \mid \Delta \vdash \mathtt{throw}\ \pi\ \mathtt{in}\ t : \sigma}\ (\textsc{Throw})$$

$$\frac{\Gamma \mid \Delta \vdash t : \tau \qquad \tilde{\sigma} \leq \mathrm{Gen}(\tau, \Gamma, \Delta) \qquad \tilde{\sigma} \succ \sigma}{\Gamma \mid \Delta \vdash (t : \tilde{\sigma}) : \sigma}\ (\textsc{Constraint-}\tilde{\sigma})$$

[7] Pour plus de détails, se reporter à [7] ou [8].

1.2.2 Typage des exceptions

On peut considérer que les structures `try-with` et `raise` sont aux exceptions ce que les structures `catch` et `throw` sont aux μ-variables. Dans cette optique, on peut rajouter un environnement de typage des exceptions Ω orthogonal aux environnements Γ et Δ et rajouter les règles de typage qui suivent pour ces structures.

$$\frac{\Gamma \mid \Delta \mid \Omega \vdash u : \sigma \qquad \Gamma \mid \Delta \mid \Omega \sqcup (C : \emptyset) \vdash t : \sigma}{\Gamma \mid \Delta \mid \Omega \vdash \texttt{try}\ t\ \texttt{with}\ C \mapsto u : \sigma}\ (\textsc{Try})$$

$$\frac{\Gamma \cup (x : \tau) \mid \Delta \mid \Omega \vdash u : \sigma \qquad \Gamma \mid \Delta \mid \Omega \sqcup (K : \mathrm{Gen}(\tau, \Gamma, \Delta, \Omega, \sigma)) \vdash t : \sigma}{\Gamma \mid \Delta \mid \Omega \vdash \texttt{try}\ t\ \texttt{with}\ Kx \mapsto u : \sigma}\ (\textsc{Try-P})$$

$$\frac{(C : \emptyset) \in \Omega}{\Gamma \mid \Delta \mid \Omega \vdash \texttt{raise}\ C : \sigma}\ (\textsc{Raise})$$

$$\frac{\Gamma \mid \Delta \mid \Omega \vdash t : \tau \qquad (K : \tilde{\tau}) \in \Omega \qquad \tilde{\tau} \succ \tau}{\Gamma \mid \Delta \mid \Omega \vdash \texttt{raise}\ Kt : \sigma}\ (\textsc{Raise-P})$$

Cependant, nous avons montré que ce point de vue était équivalent en terme de typage au point de vue de sucre syntaxique pour les exceptions. On peut donc se contenter de regarder ces règles comme des aide-mémoire et réduire le langage à un noyau minimal basé sur le couple `catch-throw`.

1.3 Sémantique opérationnelle à petits pas

On donne dans ce qui suit les règles de réduction des termes pour la sémantique à petits pas.

1.3.1 Contextes

Pour forcer l'évaluation d'un sous-terme plutôt qu'un autre, on a recours à la décomposition d'un terme en contexte à trou E et en sous-terme réductible (*redex*). On utilise les contextes conventionnels du λ-calcul en appel par nom.

La définition de la sémantique à petit pas en présence d'opérateurs est rendue complexe par la nature de la μ-substitution. Pour pouvoir conserver les propriétés du système de type, on utilise une μ-variable ω réservée *ad-hoc* qui représente la pile vide (ou de manière équivalente, le *toplevel*).

$$\begin{array}{lcl} E & ::= & [\cdot] \mid Et \mid \Lambda^{\iota}\ E\ t_1 \dots t_k \mid \Pi^{n}\ E\ f \\ H & ::= & \texttt{catch}\ \omega\ \texttt{in}\ E \end{array}$$

Intuitivement, le contexte E indique le prochain sous-terme à réduire et le *handler* H constitue l'univers extérieur d'exécution du programme.

1.3.2 Règles hors-contexte

Les règles hors-contexte sont réduites à la β-réduction au sens large.

$$\frac{}{(\texttt{fun}\ x \mapsto t)u \to_{\beta} t\langle x \leftarrow u\rangle}\ (\beta\textsc{-App})$$

$$\frac{}{\texttt{let}\ x = u\ \texttt{in}\ t \to_{\beta} t\langle x \leftarrow u\rangle}\ (\beta\textsc{-Let})$$

$$\frac{}{\texttt{let rec}\ x = u\ \texttt{in}\ t \to_{\beta} t\langle x \leftarrow \texttt{let rec}\ x = u\ \texttt{in}\ u\rangle}\ (\beta\textsc{-Let-Rec})$$

1.3.3 Primitives (δ-règles)

Les δ-règles représentent les règles de réduction primitives qui ne font pas fondalement partie du $\lambda\mu$-calcul mais qui dénotent la réduction d'extensions introduites par CTML : en l'occurrence, il s'agit de la déconstruction de types composites.

$$\begin{array}{rcl} \Pi^n\ (t_1, \ldots, t_n)\ f & \rightarrow_\delta & f t_1 \ldots t_n \\ \Lambda^\iota\ (K_j t)\ f_1 \ldots f_k & \rightarrow_\delta & f_j t \\ \Lambda^\iota\ (C_j)\ f_1 \ldots f_k & \rightarrow_\delta & f_j \\ (t : \tilde{\sigma}) & \rightarrow_\delta & t \end{array}$$

※ Pour des raisons pratiques, CTML$^+$ inclut d'autres extensions comme la gestion des entiers machine (type `int` d'Objective Caml), et les opérations de manipulation de ces objets extérieurs (opérations arithmétiques natives) sont considérées comme des δ-règles. On pourrait sans difficulté ajouter à CTML$^+$ l'équivalent de la commande `external` d'OCaml afin de définir ses propres objets abstraits et les δ-règles associées, mais cette construction n'a aucun intérêt théorique.

1.3.4 Règles contextuelles

Les premières règles induisent juste une compatibilité du contexte avec les réductions. Les autres règles sont le pendant de la règle ζ du $\lambda\mu$-calcul.

$$\frac{t \rightarrow_\beta t'}{E[t] \rightarrow_H E[t']} \qquad \frac{t \rightarrow_\delta t'}{E[t] \rightarrow_H E[t']} \qquad \frac{t \rightarrow_H t'}{H[t] \rightarrow H[t']}$$

$$\frac{}{E[\texttt{catch}\ \pi\ \texttt{in}\ t] \rightarrow_H E[t\langle \texttt{throw}\ \pi\ \texttt{in}\ u \leftarrow \texttt{throw}\ \omega\ \texttt{in}\ E[u]\rangle]}$$

$$\frac{}{E[\texttt{throw}\ \omega\ \texttt{in}\ t] \rightarrow_H t}$$

Notons que l'on pourrait décomposer la règle du `catch` en deux opérations successives, à savoir d'une part le passage au contexte sous les `throw` et d'autre part la fusion des `catch` de tête par le renommage de la variable liée en ω. C'est d'ailleurs sur cette décomposition que repose la preuve de correction.

1.4 Machine abstraite

Notre interpréteur consiste en une machine abstraite à environnement dont on donne la structure et les règles de réduction dans les paragraphes qui suivent. Elle est une adaptation à CTML de la machine décrite dans [3] et a été implémentée en OCaml.

1.4.1 Structure

Notre machine repose sur les concepts d'environnement e, de pile π et de fermeture c. On donne une définition mutuellement récursive de ces objets ci-dessous.

$$\begin{array}{rcl} e & ::= & \emptyset \mid e + (x, c) \mid e + (\alpha, \pi) \\ \pi & ::= & \varepsilon \mid c :: \pi \mid (\Pi^n t, e) :: \pi \mid (\Lambda^\iota t_1 \ldots t_k, e) :: \pi \\ c & ::= & (t, e) \end{array}$$

On donne aux environnements la sémantique attendue pour l'ajout et la récupération de contenu.

1.4.2 Réduction

La machine abstraite est décrite en termes d'états, c'est-à-dire la donnée d'un triplet terme, environnement et pile, que l'on note $\langle t, e, \pi \rangle$. Elle obéit aux règles de réduction suivantes :

$$
\begin{array}{lcrl}
\langle x, e, \pi \rangle & \to_{\mathcal{M}} & \langle t, e', \pi \rangle & e(x) = (t, e') \\
\langle tu, e, \pi \rangle & \to_{\mathcal{M}} & \langle t, e, (u, e) :: \pi \rangle & \\
\langle \lambda x.t, e, (c :: \pi) \rangle & \to_{\mathcal{M}} & \langle t, e + (x, c), \pi \rangle & \\
\langle \texttt{let } x = t \texttt{ in } u, e, \pi \rangle & \to_{\mathcal{M}} & \langle u, e + (x, (t, e)), \pi \rangle & \\
\langle \texttt{let rec } x = t \texttt{ in } u, e, \pi \rangle & \to_{\mathcal{M}} & \langle u, e + (x, (t, e')), \pi \rangle & e' = e + (x, (\texttt{let rec } x = t \texttt{ in } t, e)) \\
\langle \texttt{catch } \alpha \texttt{ in } t, e, \pi \rangle & \to_{\mathcal{M}} & \langle t, e + (\alpha, \pi), \pi \rangle & \\
\langle \texttt{throw } \alpha \texttt{ in } t, e, \pi \rangle & \to_{\mathcal{M}} & \langle t, e, \pi' \rangle & e(\alpha) = \pi' \\
\langle \Pi^n t f, e, \pi \rangle & \to_{\mathcal{M}} & \langle t, e, (\Pi^n f, e) :: \pi \rangle & \\
\langle \Lambda^\iota t f_1 \dots f_k, e, \pi \rangle & \to_{\mathcal{M}} & \langle t, e, (\Lambda^\iota f_1 \dots f_k, e) :: \pi \rangle & \\
\langle (t_1, \dots, t_n), e, (\Pi^n f, e') :: \pi \rangle & \to_{\mathcal{M}} & \langle f, e', (t_1, e) :: \dots :: (t_n, e) :: \pi \rangle & \\
\langle C_i, e, (\Lambda^\iota f_1 \dots f_k, e') :: \pi \rangle & \to_{\mathcal{M}} & \langle f_i, e', \pi \rangle & \\
\langle K_i t, e, (\Lambda^\iota f_1 \dots f_k, e') :: \pi \rangle & \to_{\mathcal{M}} & \langle f_i, e', (t, e) :: \pi \rangle &
\end{array}
$$

Les premières règles de réduction sont habituelles pour une machine abstraite réduisant le λ-calcul en appel par nom. Les règles relatives au `catch` et au `throw` caractérisent les opérations de contrôle qu'on peut réaliser : ces deux opérateurs correspondent à une manipulation explicite de la pile, sauvegarde pour `catch` et restauration pour `throw`. Les autres opérations représentent les opérations de manipulation des types composites.

1.5 Notes d'implémentation

Pour évaluer un terme, l'interpréteur procède en deux passes. Dans un premier temps, il transforme le terme en une représentation plus compacte, qu'on pourrait qualifier de *bytecode*. Il y représente les données sous forme d'entiers ou de vecteurs et applique la transformation des structures `try-with` en `catch-throw` telle que décrite précédemment[8]. La deuxième passe est la véritable application des règles de réduction de la machine précitées.

Dans CTML$^+$, le point-virgule permet de déconstruire n'importe quelle valeur d'un type quelconque et possède ainsi sa propre représentation sur la pile ; on a alors affaire à une règle similaire à la réduction des C_i sauf qu'elle est valide pour n'importe quel terme dénotant une valeur.

Puisque tel quel CTML$^+$ est un langage purement fonctionnel, il vérifie des propriétés de conservation de la forme normale par substitution de sous-termes, que ne permettent pas les langages contenant la moindre trace d'impératif. Or la β-réduction en appel par nom est coûteuse, car elle multiplie le nombre de calculs à faire par le nombre de substitutions réalisées. Prenons par exemple un terme $t \equiv (\texttt{fun } x \mapsto x + x)u$, où u est un terme qui s'évalue en un entier en de nombreuses étapes. On aura $t \to (u + u)$, d'où un doublement de la quantité de calculs à faire.

Pour améliorer l'efficacité de l'interpréteur, nous avons pensé un moment à remplacer la stratégie *call-by-name* par une stratégie *call-by-need*, qui garde trace des calculs déjà réalisés. Cependant, nous avions dans l'idée d'ajouter des constructions impératives, et nous avons abandonné le projet.

[8] Pour des raisons d'efficacité, il compacte en plus en un unique `catch` les multiples `catch` produits par la transformation en cas de déclaration de plusieurs exceptions.

2 Applications

2.1 Un classique de la logique du même

Pour ne pas déroger à la tradition, on considère le terme suivant :

$$\texttt{callcc} \equiv \texttt{fun}\ k \mapsto \texttt{catch}\ \alpha\ \texttt{in}\ k(\texttt{fun}\ x \mapsto \texttt{throw}\ \alpha\ \texttt{in}\ x)$$

Ce terme est l'opérateur `callcc` bien connu des langages de programmation fonctionnels, qui permet de manipuler explicitement la continuation courante. Dans le monde de la logique, son type est exactement la loi de Peirce classique :

$$\texttt{callcc} : ((\texttt{'a} \to \texttt{'b}) \to \texttt{'a}) \to \texttt{'a}$$

Remarquons qu'on peut l'écrire à l'aide d'un système d'exceptions à la OCaml en CTML via l'équivalence qu'on a proposé précédemment :

$$\texttt{callcc} \equiv \texttt{fun}\ k \mapsto \texttt{try}\ k(\texttt{fun}\ x \mapsto \texttt{raise}\ Ex)\ \texttt{with}\ Ex \mapsto x$$

Quoique cette syntaxe puisse être employée en OCaml, ce dernier force la définition de l'exception E avec un type fixé, ce qui fige le type de `callcc` à l'avance, alors qu'avec notre système d'exceptions ce n'est pas le cas, on peut abstraire universellement sur le type de `callcc`. Par ailleurs, comme on le verra par la suite, la sémantique de ce terme n'est pas la même en OCaml qu'en CTML.

2.2 Appel par nom et contrôle : un mariage heureux

En CTML, l'association entre d'une part une sémantique paresseuse et d'autre part des opérateurs de contrôle permet d'écrire des fonctions de méta-programmation irréalisables en OCaml, entre autres. En pratique, nous pouvons discriminer dynamiquement les fonctions strictes des autres, à l'aide de la fonction suivante :

$$\texttt{is\_strict} \equiv \texttt{fun}\ f \mapsto \texttt{catch}\ \alpha\ \texttt{in}\ f(\texttt{throw}\ \alpha\ \texttt{in}\ \texttt{true});\ \texttt{false}$$

$$\texttt{is\_strict} : (\texttt{'a} \to \texttt{'b}) \to \texttt{bool}$$

Cette fonction prend en argument une fonction f : `'a` $\to$ `'b` et renvoie `true` si cette dernière utilise son argument, `false` sinon. Elle repose sur les deux propriétés précitées. La manipulation explicite du contrôle permet d'échapper le calcul de f dès que son argument est utilisé, et l'appel par nom évite d'avoir à forcer le calcul de l'argument, ce qui provoquerait immédiatement l'échappement de la fonction.

On pourrait écrire une fonction similaire en OCaml, à l'aide des exceptions :

$$\texttt{is\_strict} \equiv \texttt{fun}\ f \mapsto \texttt{try}\ f(\texttt{raise}\ C);\ \texttt{false}\ \texttt{with}\ C \mapsto \texttt{true}$$

Cependant, cette fonction ne marchera simplement pas, par le fait même que les deux propriétés de CTML nécessaires à son bon fonctionnement sont absentes d'OCaml.

D'abord parce qu'Objective Caml est en appel par valeur, ce qui force le `raise` C à être évalué avant f, et la fonction répondrait alors toujours `true`. On pourrait outrepasser ce problème en supposant que l'argument de la fonction f doit être paresseux (type `lazy` ou apparenté[9]), avec la nouvelle définition suivante :

$$\texttt{is\_strict} \equiv \texttt{fun}\ f \mapsto \texttt{try}\ f(\texttt{fun}\ () \mapsto \texttt{raise}\ C);\ \texttt{false}\ \texttt{with}\ C \mapsto \texttt{true}$$

$$\texttt{is\_strict} : ((\texttt{'a} \to \texttt{unit}) \to \texttt{'b}) \to \texttt{bool}$$

Cependant, même sous l'hypothèse où f attend un argument paresseux, la sémantique des exceptions d'OCaml n'est pas la même. Il suffirait que f contienne une clause qui récupère l'exception C pour réduire à néant l'utilité du `try-with` de la fonction `is_strict`, comme on le verra au paragraphe suivant.

[9] En OCaml, la notation `lazy` t n'est à peine plus qu'un `fun ()` $\mapsto t$ dissimulé par un type abstrait.

2.3 Sémantique du `try-with`

La sémantique du `try-with` de CTML est substantiellement différente de celle d'OCaml. En effet, en OCaml les blocs de capture d'exceptions sont dynamiques et disparaissent une fois que le terme protégé par un `try` ne se réduit plus. On a en Objective Caml une sémantique proche de :

$$E ::= \ldots \mid \texttt{try}\ E\ \texttt{with}\ C \mapsto u$$

$$\frac{}{\texttt{try}\ v\ \texttt{with}\ C \mapsto u \to v} \qquad \frac{}{\texttt{try}\ E[\texttt{raise}\ C]\ \texttt{with}\ C \mapsto u \to u}$$

Ce n'est pas du tout le cas de CTML. En CTML, les exceptions sont définies statiquement par des liaisons, ce qui d'une part empêche au niveau du typage les exceptions non rattrapées (à l'instar des variables libres non définies), et d'autre part ne permet pas aux exceptions d'échapper leur *scope*. Prenons l'exemple simple d'une clôture qui contient une exception :

$$\texttt{exn\_escape} \equiv \texttt{try}\ (\texttt{fun}\ ()\mapsto \texttt{raise}\ C)\ \texttt{with}\ C \mapsto (\texttt{fun}\ ()\mapsto \texttt{true})$$

La différence de comportement entre OCaml et CTML est flagrante sur un tel terme.

```
exn_escape() :  Uncaught Exception : C   en OCaml
exn_escape() :  true                     en CTML
```

Une autre manifestation de cette différence est observable dans le fait qu'une exception OCaml peut être rattrapée par un bloc qui s'introduit à notre insu entre sa déclaration et sa levée. Reprenons l'exemple de `is_strict` de la section précédente, et montrons qu'encore une fois, les choses ne se passent pas du tout de la même façon dans les deux langages. On rappelle la définition de cette fonction, adaptée afin d'avoir une syntaxe identique et d'émuler un calcul paresseux :

$$\texttt{is\_strict} \equiv \texttt{fun}\ f \mapsto \texttt{try}\ f(\texttt{fun}\ ()\mapsto \texttt{raise}\ C);\ \texttt{false}\ \texttt{with}\ C \mapsto \texttt{true}$$

Prenons pour argument la fonction $f \equiv \texttt{fun}\ x \mapsto \texttt{try}\ x\texttt{()}\ \texttt{with}\ C \mapsto \texttt{()}$, et regardons ce qui se passe :

```
is_strict f :  false   en OCaml
is_strict f :  true    en CTML
```

On voit alors le problème : en OCaml, la levée d'une exception dynamique se rapporte au `try` le plus proche, alors que les exceptions statiques de CTML se rapportent exactement à la clause de rattrapage que les a définies, de manière comparable à la définition des variables par une clause `let`.

Cette propriété de conservation des exceptions est intéressante pour garantir la sécurité de l'exécution, cependant elle est assez contraignante car on s'interdit les exceptions non rattrapables. L'interpréteur adopte une position plus lâche, à savoir qu'il autorise les exceptions non déclarées (il se contente d'émettre un avertissement) mais qu'il ne peut absolument pas les rattraper.

Il est à noter que les deux systèmes d'exceptions peuvent cohabiter dans un même langage. D'un côté, les exceptions statiques du $\lambda\mu$-calcul donnent au programmeur souplesse d'utilisation (exceptions locales qu'il n'a pas à déclarer) et sécurité (exceptions toujours rattrapées) voire même efficacité[10]. De l'autre, les exceptions dynamiques de ML permettent les exceptions non-rattrapées et externes (au prix de leur déclaration et de leur type fixé), ce qui se révèle utile dans le cas d'un langage de programmation comme Objective Caml qui fait grand usage de la programmation modulaire.

Dans un esprit conservateur de la syntaxe d'OCaml, on pourrait par exemple garder la notation `try-with` et différencier lexicalement les exceptions statiques (notées par exemple comme les types variants, avec un accent à chasse fixe préfixé : `'Error`) des exceptions dynamiques (notées comme d'habitude `Error`).

[10] En effet, les exceptions statiques peuvent s'encoder par des `goto` à la compilation, tandis que les exceptions dynamiques sont posées sur la pile à l'exécution avec le lot de désagréments que cela peut entraîner : débordement de pile, entrave à la récursion terminale, etc.

3 Preuve de la correction du langage

Le typage statique fort des langages ML permet d'assurer de nombreuses propriétés de correction de leurs programmes, contrairement aux langages à typage dynamique (Python, par exemple) ou faible (au rang desquel le C) qui laissent l'utilisateur libre d'écrire des abominations pouvant être à l'origine d'anomalies allant du *glitch* à l'erreur de segmentation.

On prouvera donc dans cette partie que CTML présente les propriétés attendues d'un langage ML : stabilité et correction du typage, et correction des formes normales. On mettra ensuite en relation la sémantique à petit pas, la machine abstraite et un plongement de CTML dans le $\lambda\mu$-calcul.

Les démonstrations de la première partie s'inspirent du cours de D.E.A. de Didier Rémy [7]. Celles de la seconde partie sont à mettre en relation avec les preuves de cohérence de plongement qu'on peut trouver dans [2].

À noter que la preuve de correction du typage a tenté d'être formalisée en Coq (en s'appuyant sur des travaux tels que [1]), mais la tâche a été abandonnée faute de temps, et vu les difficultés pratiques rencontrées sur la modélisation des substitutions de la sémantique à petit pas. En effet, la notion de variable liée (et par-là même d'α-conversion) est un vrai cauchemar en preuve formelle.

Par la suite, on considèrera les termes et les schémas de types à α-équivalence près. En particulier, on pourra supposer que les variables liées d'un terme n'appartiennent pas à un ensemble choisi *a priori*. De même, pour toute substitution θ, on pourra choisir $\bar{\alpha}$ tel que $\theta(\forall\bar{\alpha}.\sigma) = \forall\bar{\alpha}.\theta(\sigma)$.

3.1 Typage

Le lemmes qui suivent constituent des propriétés essentielles de la relation de typage : cette dernière est compatible avec les opérations que l'on sera amené à faire sur les environnements de typage, à savoir substitution, extenstion et restriction.

Le lemme de substitution qui suit est fondamental, aussi bien pour prouver la correction du langage, qu'en tant qu'indicateur de ce que notre système de types n'est pas incongru.

Lemme 3.1 (Compatibilité par substitution) *Si* $\Gamma \mid \Delta \vdash t : \sigma$ *alors pour toute substitution* θ, $\theta(\Gamma) \mid \theta(\Delta) \vdash t : \theta(\sigma)$.

Preuve Par induction sur le typage de t.

1. Si $t = x$, alors $\Gamma \mid \Delta \vdash t : \sigma$ ssi $(x : \forall\bar{\alpha}.\tau) \in \Gamma$ et $\forall\bar{\alpha}.\tau \succ \sigma$. Soit ϕ une telle instanciation, on a alors $\mathrm{dom}(\phi) = \bar{\alpha}$. Quitte à α-convertir, on peut supposer que $\mathrm{dom}(\phi) \cap (\mathrm{codom}(\phi) \cup \mathrm{codom}(\theta) \cup \mathrm{dom}(\theta)) = \emptyset$. Posons $\psi(\beta) = \theta(\phi(\beta))$ si $\beta \in \bar{\alpha}$, et β sinon. Montrons que ψ instancie $\forall\bar{\alpha}.\theta(\tau)$ en $\theta(\sigma)$. Soit $\beta \in \theta(\tau)$. On distingue les différents cas d'origine de β.
   - Si $\beta \in \bar{\alpha}$, alors β n'a été ni affecté ni introduit par θ. Alors $\psi(\beta) = \theta \circ \phi(\beta)$ et cela correspond au sous-arbre de $\theta(\sigma)$ d'antécédent β par ϕ.
   - Si $\beta \notin \bar{\alpha}$ et β appartient à sous-terme introduit par θ. Alors $\psi(\beta) = \beta$ par hypothèse sur $\mathrm{codom}(\theta)$, donc en particulier ψ laisse les images de θ inchangées.
   - Si $\beta \notin \bar{\alpha}$ et β n'est pas dans un sous-terme introduit par θ. Alors $\psi(\beta) = \beta$, ce qui correspond bien à ce que l'on attend.
2. Si $t = \texttt{fun}\ x \mapsto t'$, alors $\Gamma \mid \Delta \vdash t : \sigma_1 \to \sigma_2$ ssi $\Gamma \sqcup (x : \sigma_1) \mid \Delta \vdash t' : \sigma_2$. Par induction, on tire $\theta(\Gamma) \sqcup (x : \theta(\sigma_1)) \mid \theta(\Delta) \vdash t' : \theta(\sigma_2)$ d'où $\theta(\Gamma) \mid \theta(\Delta) \vdash t : \theta(\sigma_1 \to \sigma_2)$.
3. Si $t = \texttt{let}\ x = u\ \texttt{in}\ t'$, alors $\Gamma \mid \Delta \vdash t : \sigma$ ssi $\Gamma \mid \Delta \vdash u : \tau$ et $\Gamma \sqcup (x : \forall\bar{\alpha}.\tau) \mid \Delta \vdash t' : \sigma$ où $\bar{\alpha} = \mathrm{fv}(\tau) \setminus (\mathrm{fv}(\Gamma) \cup \mathrm{fv}(\Delta))$

   Pour tout $\alpha \in \bar{\alpha}$, soit $\alpha' \notin (\mathrm{dom}(\theta) \cup \mathrm{fv}(\Gamma) \cup \mathrm{fv}(\Delta))$. Soit ϕ qui à tout α associe α'. Posons $\hat{\theta} = \theta \circ \phi$.

   Par hypothèse d'induction, on tire :

$$\begin{array}{rcl} \hat{\theta}(\Gamma) & \mid & \hat{\theta}(\Delta) \vdash u : \hat{\theta}(\tau) \\ \theta(\Gamma) \sqcup (x : \theta(\forall\bar{\alpha}.\tau)) & \mid & \theta(\Delta) \vdash t' : \theta(\sigma) \end{array}$$

Vu comme on a choisi $\hat{\theta}$, on a alors $\hat{\theta}(\Gamma) = \theta(\Gamma)$ et $\hat{\theta}(\Delta) = \theta(\Delta)$, et d'autre part $\theta(\forall\bar{\alpha}.\tau) = \forall\bar{\alpha}'.\hat{\theta}(\tau)$. Ainsi :

$$\begin{array}{rcl} \theta(\Gamma) & \mid & \theta(\Delta) \vdash u : \hat{\theta}(\tau) \\ \theta(\Gamma) \sqcup (x : \forall\bar{\alpha}'.\hat{\theta}(\tau)) & \mid & \theta(\Delta) \vdash t' : \theta(\sigma) \end{array}$$

Remarquons alors que $\bar{\alpha}' \subseteq \mathrm{fv}(\hat{\theta}(\tau)) \setminus (\mathrm{fv}(\theta(\Gamma)) \cup \mathrm{fv}(\theta(\Delta)))$. Le schéma $\mathrm{Gen}(\hat{\theta}(\tau), \theta(\Gamma), \theta(\Delta))$ est donc plus général que $\forall\bar{\alpha}'.\hat{\theta}(\tau)$ et par conséquent :

$$\theta(\Gamma) \sqcup (x : \mathrm{Gen}(\hat{\theta}(\tau), \theta(\Gamma), \theta(\Delta))) \mid \theta(\Delta) \vdash u : \theta(\sigma)$$

On conclut alors par la règle de typage du LET.

4. Si $t \equiv \texttt{let rec}\ x = u\ \texttt{in}\ t'$, on applique le même raisonnement que précédemment.
5. Les autres cas se font de manière directe en appliquant l'hypothèse d'induction sur le sous-terme, puisqu'il n'y a pas de généralisation.

Les deux lemmes suivants permettent d'affirmer qu'ajouter ou retirer des informations non utilisées par la preuve du typage d'un terme n'affectent en rien le résultat de la dérivation. Les preuves, plutôt ennuyeuses, se font assez aisément par induction sur la relation de typage. Il est à noter que la preuve du lemme d'extension n'est pas si évidente, car il faut faire appel au lemme de substitution qui précède dans les cas où il y a généralisation du type.

Lemme 3.2 (Extension) *Pour tous environnements $\Gamma \subseteq \tilde{\Gamma}$, $\Delta \subseteq \tilde{\Delta}$, si $\Gamma \mid \Delta \vdash t : \sigma$, alors $\tilde{\Gamma} \mid \tilde{\Delta} \vdash t : \sigma$.*

Lemme 3.3 (Restriction) *Pour tous environnements Γ, Δ, si $\Gamma \mid \Delta \vdash t : \sigma$, alors $\Gamma \cap \lambda fv(t) \mid \Delta \cap \mu fv(t) \vdash t : \sigma$.*

3.2 Sémantique

On montrera dans cette partie la compatibilité entre typage (notion statique) et réduction (notion dynamique), plus connue sous son appellation anglophone de *subject reduction*, propriété qui justifie le fort typage des langages ML.

On définit maintenant une relation d'ordre (partielle) « au moins aussi typable » sur les termes, qui nous servira par la suite pour décrire la typabilité d'un terme par rapport à un autre.

Définition 3.1 *Pour tous termes t_1, t_2 on note $t_1 \sqsubseteq t_2$ si pour tous Γ, Δ, σ, si $\Gamma \mid \Delta \vdash t_1 : \sigma$ alors $\Gamma \mid \Delta \vdash t_2 : \sigma$. On notera $t_1 \cong t_2$ si $t_1 \sqsubseteq t_2$ et $t_2 \sqsubseteq t_1$.*

Pour tous termes t_1, t_2 on note $t_1 \sqsubseteq_\omega t_2$ si pour tous Γ, Δ, σ, si $\Gamma \mid \Delta \cup (\omega : \sigma) \vdash t_1 : \sigma$ alors $\Gamma \mid \Delta \cup (\omega : \sigma) \vdash t_2 : \sigma$.

Lemme 3.4 *On a la relation suivante : $(\sqsubseteq_\omega) \subseteq (\sqsubseteq)$.*

3.2.1 Correction des primitives

Définition 3.2 *On notera Γ_0 un environnement initial (ici, vide).*

Le lemme qui suit introduit la compabilité du typage avec les opérations primitives de CTML externes au $\lambda\mu$-calcul. Il est nécessaire pour s'assurer que ces primitives ne sont la source ni d'erreurs de typage, ni de blocage anormal de l'évaluation.

Lemme 3.5 (Correction des δ-règles) *Les δ-règles vérifient les propriétés suivantes :*

H1 *Si $t \to_\delta t'$ alors $t \sqsubseteq t'$.*

H2 *Si $t = \Lambda^\iota v f_1 \dots f_k$ ou $t = \Pi^n v f$ ou $t = (t' : \tilde{\sigma})$ et t est typable dans l'environnement initial, alors il existe un et un seul terme t' tel que $t \to_\delta t'$.*

Preuve On a seulement trois types de δ-règles : destruction des tuples, des types sommes et de la contrainte de type. Chacune est déterministe, donc la deuxième propriété est vérifiée.

La première propriété est vérifiée par construction des règles de typage pour les types produits et sommes et pour la contrainte de type.

Nous tenons à souligner que l'on peut étendre Γ_0 à tout environnement typant exactement les fonctions externes éventuelles du langage[11], sous réserve que les celles-ci vérifient toujours le lemme de compatibilité des δ-règles.

3.2.2 Déterminisme

Il est utile de remarquer que la réduction de CTML est déterministe ; cette propriété sera utile par la suite.

Lemme 3.6 (Unicité du contexte) *Pour tout terme t, il existe un unique contexte E et un unique terme u tel que $t = E[u]$ et u ne commence pas par un contexte. On appellera E* contexte de tête.

Preuve Par induction sur la structure des termes.

Théorème 3.7 (Déterminisme) *La réduction $\to$ est déterministe.*

Preuve On regarde le contexte de tête de t qui est unique par le lemme précédent. Il suffit donc de montrer que la réduction sous le contexte est déterministe.

1. Si $t = x$, $t \not\to$.
2. Si $t = C$, $t \not\to$.
3. Si $t = Kt'$, $t \not\to$.
4. Si $t = (t_1, \dots, t_n)$, $t \not\to$.
5. Si $t = \texttt{fun}\ x \mapsto t'$, $t \not\to$.
6. Si $t = \texttt{throw}\ \pi\ \texttt{in}\ t'$, $t \not\to$.
7. Si $t = \texttt{throw}\ \omega\ \texttt{in}\ t'$, alors il n'y a qu'une seule réduction.
8. Si $t = \texttt{let}\ x = u\ \texttt{in}\ t'$, alors il n'y a qu'une seule réduction et $t \to t'\langle x \leftarrow u\rangle$.
9. Si $t = \texttt{let rec}\ x = u\ \texttt{in}\ t'$, alors il n'y a qu'une seule réduction et $t \to t'\langle x \leftarrow \texttt{let rec}\ x = u\ \texttt{in}\ u\rangle$.
10. Si $t = t_1 t_2$, et que la règle du contexte ne s'applique pas, alors il y a trois sous-cas :
    (a) Soit $t_1 = \texttt{fun}\ x \mapsto u_1$, auquel cas c'est une valeur, seule la règle de substitution s'applique.
    (b) Soit t_1 est une valeur non-fonctionnelle, auquel cas $t \not\to$.

[11] Comme par exemple les opérations arithmétiques de CTML$^+$.

(c) Soit t_1 n'est pas une valeur, donc l'évaluation est bloquée et $t \not\to$.

11. Si t est un déconstructeur et que la règle du contexte ne s'applique pas, alors il y a trois sous-cas :
    (a) Soit t_1 est bien une valeur du type attendu par le déconstructeur, dans ce cas par hypothèse il existe une δ-règle qui s'applique et elle est unique.
    (b) Soit t_1 est une valeur d'un type autre, dans ce cas l'évaluation se bloque et $t \not\to$.
    (c) Soit t_1 n'est pas une valeur, idem.
12. Si $t = \texttt{catch}\ \pi\ \texttt{in}\ t'$ alors il n'y a qu'une seule règle qui s'applique.

Remarquons ici que le déterminisme n'est pas une propriété intrinsèque aux langages ML[12]. Cependant, elle facilite énormément certaines preuves à venir[13]. Comme CTML est par nature déterministe, il n'y a pas lieu de se priver de cette facilité.

3.2.3 Préservation du typage par la réduction

Nous voici arrivés au gros de la preuve de la correction du typage CTML. Le théorème ci-dessous est sans doute le résultat fondamental de ce langage. Conjoint à la preuve de correction des formes normales qu'on verra par la suite, il nous assure que l'évaluation d'un terme ne va pas exploser en vol.

Théorème 3.8 (*Subject reduction*) *Si $H[t] \to H[t']$ alors $H[t] \sqsubseteq H[t']$.*

Nous ne rentrerons pas ici dans les détails de la preuve, car elle s'obtient directement en mettant bout à bout les lemmes qui suivent, et ne consisterait qu'en une analyse peu intéressante au cas par cas des règles de réduction. Ce théorème repose en fait sur trois lemmes fondamentaux qui correspondent aux trois types de réduction possible :

1. compatibilité du typage avec la β-réduction pour les λ-variables (lemme 3.11) ;
2. compatibilité du typage avec la ζ-réduction pour les μ-variables (lemme 3.14) ;
3. compatibilité du typage avec la δ-réduction pour les primitives (lemme 3.5).

Si on leur associe les lemmes de passage au contexte et au *handler* ci-dessous, le résultat est immédiat.

Lemme 3.9 (Compatibilité du handler) *Pour tous termes t, t', $t \sqsubseteq_\omega t'$ ssi $H[t] \sqsubseteq H[t']$.*

Preuve Il suffit d'appliquer la règle CATCH.

Lemme 3.10 (Compatibilité du contexte) *Pour tous termes t, t', pour tout contexte E, si $t \sqsubseteq t'$ alors $E[t] \sqsubseteq E[t']$.*

Preuve Par induction sur le contexte. Soient Γ, Δ des environnements et σ un type tels que $\Gamma \mid \Delta \vdash E[t] : \sigma$. Soit t' tel que $t \sqsubseteq t'$.

1. Si $E = [\cdot]$, c'est évident.
2. Si $E = E'u$ alors l'arbre de typage de $E[t]$ se termine par la règle APP, c'est-à-dire qu'il existe τ tel que $\Gamma \mid \Delta \vdash E'[t] : \tau \to \sigma$ et $\Gamma \mid \Delta \vdash u : \tau$.
   Par hypothèse de récurrence $\Gamma \mid \Delta \vdash E'[t'] : \tau \to \sigma$, donc on peut appliquer la règle APP et on tire $\Gamma \mid \Delta \vdash E'[t']u : \sigma$.
3. Si E est un déconstructeur, on utilise une méthode similaire en appliquant l'hypothèse d'induction.

Lemme 3.11 (Compatibilité de la λ-substitution) *Si $\Gamma \mid \Delta \vdash u : \tau$ et $\Gamma \sqcup (x : \forall\bar{\alpha}.\tau) \mid \Delta \vdash t : \sigma$, si les $\bar{\alpha}$ ne sont pas libres dans $\Gamma \cup \Delta \cup \{\sigma\}$, alors $\Gamma \mid \Delta \vdash t\langle x \leftarrow u\rangle : \sigma$.*

Corollaire 3.12 *Si $t \to_\beta t'$ alors $t \sqsubseteq t'$.*

Preuve Par induction sur le typage t. Les seuls cas intéressants sont les cas de base et les liaisons.

[12] Ne serait-ce que la spécification d'Objective Caml qui explique posément que, contrairement à ce qu'on pourrait imaginer, ce langage n'est *pas* déterministe. Cf. pour plus de détails la partie sémantique opérationnelle de [8].

[13] Elle évite d'avoir à prouver la confluence de la réduction.

Si $t = x$: Dans ce cas $t\langle x \leftarrow u\rangle = u$. D'après la règle VAR, $\forall\bar{\alpha}.\tau \succ \sigma$. Soit θ cette instanciation, alors en particulier θ laisse les environnements inchangés (par hypothèse, $\bar{\alpha}$ n'est pas libre dans les environnements) et par le lemme 3.1 on a le résultat.

Si $t = y$ et $y \neq x$: Dans ce cas x n'est pas libre dans t, et toute substitution de x laisse t inchangé. L'hypothèse sur x n'est donc pas nécessaire. On a le résultat en restreignant $\Gamma \sqcup (x : \forall\bar{\alpha}.\tau)$ à Γ par le lemme 3.3.

Si $t = \texttt{fun}\ y \mapsto t'$ avec $y \neq x$: D'après la règle FUN, σ est de la forme $\sigma_1 \to \sigma_2$ on a la dérivation :

$$\Gamma \sqcup (y : \sigma_1) \sqcup (x : \forall\bar{\alpha}.\tau) \mid \Delta \vdash t' : \sigma_2$$

Par hypothèse, les $\bar{\alpha}$ ne sont pas libres dans σ, donc dans σ_1 en particulier. Les hypothèses sur $\bar{\alpha}$ restent vérifiées. On applique l'hypothèse de récurrence sur t' et on obtient le résultat.

Si $t = \texttt{let}\ y = e\ \texttt{in}\ t'$ avec $y \neq x$: La règle LET nous assure les dérivations suivantes :

$$\Gamma \sqcup (x : \forall\bar{\alpha}.\tau) \mid \Delta \vdash e : \upsilon$$
$$\Gamma \sqcup (y : \mathrm{Gen}(\upsilon, \Gamma, \Delta)) \sqcup (x : \forall\bar{\alpha}.\tau) \mid \Delta \vdash t' : \sigma$$

On applique l'hypothèse de récurrence à e puis à t', puisqu'en particulier $\bar{\alpha}$ n'est libre ni dans υ (si c'était le cas, on pourrait renommer les variables libre de υ par le lemme 3.1) ni dans σ. En appliquant la règle du LET, on retrouve le résultat attendu.

Si $t = \texttt{let rec}\ y = e\ \texttt{in}\ t'$ et $y \neq x$: On démontre la propriété de manière analogue au cas du `let`.

Autres cas : On se contente d'appliquer l'hypothèse de récurrence sur les sous-termes.

Lemme 3.13 (Destruction du contexte) *S'il existe τ tel que $\Gamma \mid \Delta \cup (\pi : \sigma) \vdash E[\texttt{throw}\ \pi\ \texttt{in}\ u] : \tau$, alors $\Gamma \mid \Delta \cup (\pi : \sigma) \vdash u : \sigma$.*

Preuve La preuve se fait aisément par induction sur le typage de $E[\texttt{throw}\ \pi\ \texttt{in}\ u]$.

Lemme 3.14 (Compatibilité de la μ-substitution) *Pour tout t, pour tout E tel que $\pi \notin fv(E)$, si $\Gamma \sqcup (x : \sigma) \mid \Delta \vdash E[x] : \tau$ pour $x \notin fv(E) \cup fv(\Gamma)$ et si $\Gamma \mid \Delta \sqcup (\pi : \sigma) \vdash t : \upsilon$ alors $\Gamma \mid \Delta \sqcup (\pi : \tau) \vdash t\langle\texttt{throw}\ \pi\ \texttt{in}\ u \leftarrow \texttt{throw}\ \pi\ \texttt{in}\ E[u]\rangle : \upsilon$.*

Preuve Par induction sur le typage de t. On notera $t\langle\pi \leftarrow E\rangle$ pour $t\langle\texttt{throw}\ \pi\ \texttt{in}\ u \leftarrow \texttt{throw}\ \pi\ \texttt{in}\ E[u]\rangle$.

Si $t = x$ ou $t = C$. Comme $\pi \notin \mathrm{fv}(t)$, la substitution laisse t inchangé et on peut appliquer successivement le lemme de restriction puis le lemme d'extension à π pour avoir le résultat.

Si $t = \texttt{throw}\ \pi\ \texttt{in}\ t'$. Par la règle THROW, on a $\Gamma \mid \Delta \sqcup (\pi : \sigma) \vdash t' : \sigma$. Par hypothèse d'induction, on tire $\Gamma \mid \Delta \sqcup (\pi : \tau) \vdash t'\langle\pi \leftarrow E\rangle : \sigma$

Alors en particulier on a $\Gamma \mid \Delta \sqcup (\pi : \tau) \vdash E[t'\langle\pi \leftarrow E\rangle] : \tau$ d'après les hypothèses faites sur E. On applique la règle THROW et on tire le résultat.

Si $t = \texttt{throw}\ \rho\ \texttt{in}\ t'$ et $\rho \neq \pi$. Par la règle THROW, on a $\Gamma \mid \Delta \sqcup (\pi : \sigma) \vdash t' : \upsilon'$ et $(\rho : \upsilon') \in \Delta$. Par hypothèse d'induction, on tire $\Gamma \mid \Delta \sqcup (\pi : \tau) \vdash t'\langle\pi \leftarrow E\rangle : \upsilon'$. On réapplique la règle THROW et on a le résultat.

Si $t = \texttt{catch}\ \rho\ \texttt{in}\ t'$ avec $\rho \neq \pi$. La règle CATCH nous assure que $\Gamma \mid \Delta \sqcup (\rho : \upsilon) \sqcup (\pi : \sigma) \vdash t' : \upsilon$. On peut supposer, quitte à α-convertir, qu'on a $\rho \notin \mathrm{fv}(E)$. Par le lemme d'extension d'environnement (3.2), on a alors $\Gamma \sqcup (x : \sigma) \mid \Delta \sqcup (\rho : \upsilon) \vdash E[x] : \tau$ pour $x \notin \mathrm{fv}(E) \cup \mathrm{fv}(\Gamma)$. On conclut par l'hypothèse de récurrence et la règle CATCH.

Si $t = \texttt{fun}\ x \mapsto t'$. D'après la règle FUN, on a $\Gamma \sqcup (x : \upsilon_1) \mid \Delta \sqcup (\pi : \sigma) \vdash t' : \upsilon_2$. Quitte à α-convertir, on a $x \notin \mathrm{fv}(E)$ et par le lemme d'extension d'environnement, on a aussi $\Gamma \sqcup (x : \upsilon_1) \mid \Delta \vdash E[y] : \tau$ pour $y \notin \mathrm{fv}(E) \cup \mathrm{fv}(\Gamma)$. On conclut par l'hypothèse de récurrence et la règle FUN.

Autres cas. On applique l'hypothèse d'induction.

Lemme 3.15 (Compatibilité du `catch`) *Pour tout t, pour tout E tel que $\pi \notin fv(E)$ on a :*

$$E[\texttt{catch}\ \pi\ \texttt{in}\ t] \sqsubseteq \texttt{catch}\ \pi\ \texttt{in}\ E[t\langle \texttt{throw}\ \pi\ \texttt{in}\ u \leftarrow \texttt{throw}\ \pi\ \texttt{in}\ E[u]\rangle]$$

Preuve Par induction sur le typage de E.

Si $E = [\cdot]$. Alors le résultat est immédiat, les deux termes sont égaux.

Si $E = E'w$. Par la règle APP, on a alors d'une part $\Gamma \mid \Delta \vdash E'[\texttt{catch}\ \pi\ \texttt{in}\ t] : \tau \to \sigma$ et d'autre part $\Gamma \mid \Delta \vdash w : \tau$.

L'hypothèse d'induction appliquée à la première dérivation nous fournit :

$$\Gamma \mid \Delta \vdash \texttt{catch}\ \pi\ \texttt{in}\ E'[t\langle \pi \leftarrow E'\rangle] : \tau \to \sigma$$

D'où, par la règle CATCH, on tire :

$$\Gamma \mid \Delta \sqcup (\pi : \tau \to \sigma) \vdash E'[t\langle \pi \leftarrow E'\rangle] : \tau \to \sigma$$

Comme $\pi \notin \mathrm{fv}(E)$, le lemme de compatibilité de la μ-substitution et le typage de E nous assurent alors :

$$\Gamma \mid \Delta \sqcup (\pi : \sigma) \vdash E'[t\langle \pi \leftarrow E\rangle] : \tau \to \sigma$$

On conclut en appliquant successivement la règle APP puis la règle CATCH.

Autres cas. Ils sont traités de manière analogue au cas précédent.

Lemme 3.16 (Compatibilité du `throw`) *Pour tout E, pour tout t, pour tout π on a :*

$$\texttt{catch}\ \pi\ \texttt{in}\ E[\texttt{throw}\ \pi\ \texttt{in}\ t] \sqsubseteq \texttt{catch}\ \pi\ \texttt{in}\ t$$

Preuve Si $\Gamma \mid \Delta \vdash \texttt{catch}\ \pi\ \texttt{in}\ E[\texttt{throw}\ \pi\ \texttt{in}\ t] : \sigma$, alors d'après la règle du CATCH, $\Gamma \mid \Delta \cup (\pi : \sigma) \vdash E[\texttt{throw}\ \pi\ \texttt{in}\ t] : \sigma$. On conclut par le lemme de destruction du contexte.

3.2.4 Correction des formes normales

On montre ici la correction de la réduction, à savoir que les formes normales de la réduction des termes bien typés de CTML ont bien la forme attendue par leur type. Comme énoncé précédemment, ce théorème allié à celui de *subject reduction* nous assure qu'un programme bien typé se comporte correctement.

Lemme 3.17 *Si $\Gamma_0 \mid (\omega : \upsilon) \vdash t : \sigma$ (ce que l'on note t Γ_0-typable de type σ) et si la réduction de $H[t]$ termine, alors elle termine sur un terme de la forme $H[v]$ où v est une valeur de la forme :*

- *C ou Kt si $\sigma = \iota\{\bar{\tau}\}$;*
- *$(t_1, \ldots, t_n)$ si $\sigma = (\sigma_1 * \ldots * \sigma_n)$;*
- *$\texttt{fun}\ x \mapsto t'$ si $\sigma = \sigma_1 \to \sigma_2$.*

Preuve On regarde les cas où un terme t pourrait se bloquer alors qu'il n'est pas une valeur telle qu'attendue, et on raisonne par induction.

Si $t = x$. Dans ce cas $x \notin \Gamma_0$ donc en particulier, t n'est pas Γ_0-typable.

Si $t = C$ ou $t = Kt'$. Par hypothèse de bon typage, ce terme est une valeur de la forme attendue, et il est en forme normale.

Si $t = (t_1, \ldots, t_n)$. Idem.

Si $t = \texttt{fun}\ x \mapsto t'$. Idem.

Si $t = t_1 t_2$. Supposons t Γ_0-typable de type σ. Par la règle App, t_1 est Γ_0-typable de type $\tau \to \sigma$ pour un certain τ. Or t ne se réduit pas selon la règle β, sinon ce ne serait pas une forme normale. Par conséquent $t_1 \neq \texttt{fun}\ x \mapsto t'_1$. Pourtant t_1 ne se réduit pas non plus, sinon on pourrait appliquer la règle d'environnement. Cela nie l'hypothèse de récurrence, c'est donc absurde.

Si $t = \texttt{let}\ x = u\ \texttt{in}\ t'$ ou $t = \texttt{let rec}\ x = u\ \texttt{in}\ t'$. De tels termes se réduisent selon une des règles β et ne sont donc pas des formes normales.

Si t est un déconstructeur sur t'. De deux choses l'une. Soit t' ne se réduit pas, et par hypothèse de récurrence c'est une valeur de la forme et du type attendus. Dans ce cas, ces termes se réduisent par les δ-règles d'après **H2**, et c'est absurde. Soit t' se réduit, et par passage au contexte t se réduit aussi : absurde de même.

Si $t = \texttt{catch}\ \pi\ \texttt{in}\ t'$. Ce terme n'est pas une forme normale par la règle de réduction du `catch`.

Si $t = \texttt{throw}\ \pi\ \texttt{in}\ t'$. Ce terme n'est pas Γ_0-typable si $\pi \neq \omega$, et il se réduit sinon. Dans les deux cas c'est absurde.

Théorème 3.18 *Si $\Gamma_0 \mid \emptyset \vdash H[t] : \sigma$ alors de deux choses l'une :*

1. *Soit la réduction de $H[t]$ boucle ;*
2. *Soit elle s'arrête sur un terme de la forme $H[v]$, avec v une valeur de la forme :*
   - *C ou Kt si $\sigma = \iota\{\bar{\tau}\}$;*
   - *$(t_1, \ldots, t_n)$ si $\sigma = (\sigma_1 * \ldots * \sigma_n)$;*
   - *$\texttt{fun}\ x \mapsto t'$ si $\sigma = \sigma_1 \to \sigma_2$.*

Preuve C'est un cas particulier du lemme précédent.

3.2.5 Machine abstraite

On s'attache dans cette partie à montrer un invariant qui lie la sémantique opérationnelle à petit pas et la réduction de la machine abstraite, avec pour finalité la preuve de la correction sémantique de la machine abstraite. Le point de vue adopté ici repose sur une mise en relation des termes de CTML avec les états de la machine abstraite.

On aurait pu, dans un point de vue complètement orthogonal, considérer un typage des états de la machine, comme décrit dans [3], mais cette étude serait redondante et aurait un développement analogue à celui de la partie précédente.

Définition 3.3 *On définit par induction une traduction $\{\!\{\}\!\}^\lambda$ des clôtures de la machine abstraite vers les termes de CTML et une traduction $\{\!\{\}\!\}^\mu$ des piles de la machine abstraite vers les contextes de CTML.*

- $\{\!\{x\}\!\}^\lambda_e = \{\!\{t\}\!\}^\lambda_{e'}$ *si* $e(x) = (t, e')$;
- $\{\!\{x\}\!\}^\lambda_e = x$ *si* $x \notin e$;
- $\{\!\{C\}\!\}^\lambda_e = C$;

- $\{\!\{Kt\}\!\}^\lambda_e = K\{\!\{t\}\!\}^\lambda_e$;
- $\{\!\{(t_1, \ldots, t_n)\}\!\}^\lambda_e = (\{\!\{t_1\}\!\}^\lambda_e, \ldots, \{\!\{t_n\}\!\}^\lambda_e)$;
- $\{\!\{\texttt{fun}\ x \mapsto t\}\!\}^\lambda_e = \texttt{fun}\ x \mapsto \{\!\{t\}\!\}^\lambda_e$ *avec* $x \notin fv(e)$;
- $\{\!\{t_1 t_2\}\!\}^\lambda_e = \{\!\{t_1\}\!\}^\lambda_e \{\!\{t_2\}\!\}^\lambda_e$;
- $\{\!\{\texttt{let}\ x = u\ \texttt{in}\ t\}\!\}^\lambda_e = \texttt{let}\ x = \{\!\{u\}\!\}^\lambda_e\ \texttt{in}\ \{\!\{t\}\!\}^\lambda_e$ *avec* $x \notin fv(e)$;
- $\{\!\{\texttt{let rec}\ x = u\ \texttt{in}\ t\}\!\}^\lambda_e = \texttt{let rec}\ x = \{\!\{u\}\!\}^\lambda_e\ \texttt{in}\ \{\!\{t\}\!\}^\lambda_e$ *avec* $x \notin fv(e)$;
- $\{\!\{\Pi^n t f\}\!\}^\lambda_e = \Pi^n \{\!\{t\}\!\}^\lambda_e \{\!\{f\}\!\}^\lambda_e$;
- $\{\!\{\Lambda^\iota t f_1 \ldots f_k\}\!\}^\lambda_e = \Lambda^\iota \{\!\{t\}\!\}^\lambda_e \{\!\{f_1\}\!\}^\lambda_e \ldots \{\!\{f_k\}\!\}^\lambda_e$;
- $\{\!\{\texttt{catch}\ \alpha\ \texttt{in}\ t\}\!\}^\lambda_e = \texttt{catch}\ \alpha\ \texttt{in}\ \{\!\{t\}\!\}^\lambda_e$ *avec* $\alpha \notin fv(e)$;
- $\{\!\{\texttt{throw}\ \alpha\ \texttt{in}\ t\}\!\}^\lambda_e = \texttt{throw}\ \omega\ \texttt{in}\ \{\!\{\pi\}\!\}^\mu[\{\!\{t\}\!\}^\lambda_e]$ *si* $e(\alpha) = \pi$;
- $\{\!\{\texttt{throw}\ \alpha\ \texttt{in}\ t\}\!\}^\lambda_e = \texttt{throw}\ \alpha\ \texttt{in}\ \{\!\{t\}\!\}^\lambda_e$ *si* $\alpha \notin e$;
- $\{\!\{(t : \tilde{\sigma})\}\!\}^\lambda_e = (\{\!\{t\}\!\}^\lambda_e : \tilde{\sigma})$.

- $\{\!\{\varepsilon\}\!\}^\mu = [\cdot]$;
- $\{\!\{(t, e) :: \pi\}\!\}^\mu = \{\!\{\pi\}\!\}^\mu[[\cdot]\{\!\{t\}\!\}^\lambda_e]$;
- $\{\!\{(\Pi^n f, e) :: \pi\}\!\}^\mu = \{\!\{\pi\}\!\}^\mu[\Pi^n[\cdot]\{\!\{f\}\!\}^\lambda_e]$;
- $\{\!\{(\Lambda^\iota f_1 f_k, e) :: \pi\}\!\}^\mu = \{\!\{\pi\}\!\}^\mu[\Lambda^\iota[\cdot]\{\!\{f_1\}\!\}^\lambda_e \ldots \{\!\{f_k\}\!\}^\lambda_e]$;

Cette définition est bien fondée car par définition inductive des environnements, dans l'égalité $\{\!\{x\}\!\}^\lambda_e = \{\!\{t\}\!\}^\lambda_{e'}$ *avec* $e(x) = (t, e')$*, on a* $e' \leq e$ *pour l'ordre induit par la structure des environnements, et dans les autres cas c'est le terme qui décroît. Par conséquent, cette réécriture termine et la traduction est bien définie.*

Enfin, on définit une traduction $\{\!\{\}\!\}$ *d'un état de la machine abstraite vers un terme de* CTML *comme* $\{\!\{\langle t, e, \pi\rangle\}\!\} = H[\{\!\{\pi\}\!\}^\mu[\{\!\{t\}\!\}^\lambda_e]]$.

Nous remarquerons avec attention la différence de traitement entre les variables apparaissant dans l'environnement et celles qui n'y apparaissent pas. Elle réside dans le fait qu'un état de la machine abstraite est une entité dynamique, qui applique les substitutions paresseusement (son environnement contient exactement les substitutions *à faire*) alors que la sémantique opérationnelle procède par réécriture complète à chaque substitution.

Ce concept est formalisé dans les deux lemmes qui suivent :

Lemme 3.19 (Équivalence de la λ-substitution) *Pour tout terme* t*, pour tout environnement* e*, pour toute clôture* (u, e') *on a* $\{\!\{t\}\!\}^\lambda_{e+(x=(u,e'))} = \{\!\{t\}\!\}^\lambda_e \langle x \leftarrow \{\!\{u\}\!\}^\lambda_{e'}\rangle$.

Lemme 3.20 (Équivalence de la μ-substitution) *Pour tout terme* t*, pour tout environnement* e*, pour toute pile* π *on a* $\{\!\{t\}\!\}^\lambda_{e+(\alpha=\pi)} = \{\!\{t\}\!\}^\lambda_e \langle \alpha \leftarrow \{\!\{\pi\}\!\}^\mu\rangle$.

On montre maintenant que notre traduction est compatible avec la sémantique opérationnelle, d'une part en ne se réduisant pas plus que la clôture réflexive de $\rightarrow$, d'autre part en ne se bloquant pas plus non plus.

Lemme 3.21 (Conservation de la sémantique) *Soit* s *un état de la machine abstraite. Si* $s \rightarrow_{\mathcal{M}} s'$*, alors* $\{\!\{s\}\!\} \rightarrow \{\!\{s'\}\!\}$ *ou* $\{\!\{s\}\!\} = \{\!\{s'\}\!\}$.

Corollaire 3.22 *Si* $s \overset{*}{\rightarrow}_{\mathcal{M}} s'$*, alors* $\{\!\{s\}\!\} \overset{*}{\rightarrow} \{\!\{s'\}\!\}$.

Preuve Par induction sur les règles de réduction de la machine.

Si $s = \langle x, e, \pi\rangle$ **et** $e(x) = (t, e')$**.** Alors $s' = \langle t, e', \pi\rangle$. Par définition de $\{\!\{\}\!\}$, on a $\{\!\{s\}\!\} = \{\!\{s'\}\!\}$.

Si $s = \langle tu, e, \pi\rangle$**.** Alors $s' = \langle t, e, (u, e) :: \pi\rangle$. Alors $\{\!\{s\}\!\} = H[\{\!\{\pi\}\!\}^\mu[\{\!\{t\}\!\}^\lambda_e \{\!\{u\}\!\}^\lambda_e]]$ et $\{\!\{s'\}\!\} = H[\{\!\{(u, e) :: \pi\}\!\}^\mu[\{\!\{t\}\!\}^\lambda_e]] = H[\{\!\{\pi\}\!\}^\mu[\{\!\{t\}\!\}^\lambda_e \{\!\{u\}\!\}^\lambda_e]]$. Donc $\{\!\{s\}\!\} = \{\!\{s'\}\!\}$.

Si $s = \langle \texttt{fun}\ x \mapsto t, e, (u, e') :: \pi \rangle$**.** Alors $s' = \langle t, e + (x = (u, e')), \pi \rangle$. On a $\{\!\{s\}\!\} = H[\{\!\{(u, e') :: \pi\}\!\}^{\mu}[\texttt{fun}\ x \mapsto \{\!\{t\}\!\}^{\lambda}_{e}]] = H[\{\!\{\pi\}\!\}^{\mu}[(\texttt{fun}\ x \mapsto \{\!\{t\}\!\}^{\lambda}_{e})\{\!\{u\}\!\}^{\lambda}_{e'}]]$. D'autre part $\{\!\{s'\}\!\} = H[\{\!\{\pi\}\!\}^{\mu}[\{\!\{t\}\!\}^{\lambda}_{e+(x=(u,e'))}]] = H[\{\!\{\pi\}\!\}^{\mu}[\{\!\{t\}\!\}^{\lambda}_{e}\langle x \leftarrow \{\!\{u\}\!\}^{\lambda}_{e'}\rangle]]$. Donc $\{\!\{s\}\!\} \to \{\!\{s'\}\!\}$ par la règle de β-réduction.

Si le symbole de tête du terme est un $\texttt{let}\ x = u\ \texttt{in}\ t$ **ou un** $\texttt{let rec}\ x = u\ \texttt{in}\ t$**.** Alors on raisonne de manière similaire au point précédent.

Si $s = \langle \texttt{catch}\ \alpha\ \texttt{in}\ t, e, \pi \rangle$**.** Alors $s' = \langle t, e + (\alpha = \pi), \pi \rangle$. On a $\{\!\{s\}\!\} = H[\{\!\{\pi\}\!\}^{\mu}[\texttt{catch}\ \alpha\ \texttt{in}\ \{\!\{t\}\!\}^{\lambda}_{e}]]$. D'autre part $\{\!\{s'\}\!\} = H[\{\!\{\pi\}\!\}^{\mu}[\{\!\{t\}\!\}^{\lambda}_{e+(\alpha=\pi)}]] = H[\{\!\{\pi\}\!\}^{\mu}\{\!\{t\}\!\}^{\lambda}_{e}\langle \alpha \leftarrow \{\!\{\pi\}\!\}^{\mu}\rangle]$. D'où $\{\!\{s\}\!\} \to \{\!\{s'\}\!\}$.

Si $s = \langle \texttt{throw}\ \alpha\ \texttt{in}\ t, e, \pi \rangle$ **et** $e(\alpha) = \rho$**.** Alors $s' = \langle t, e, \rho \rangle$. On a $\{\!\{s\}\!\} = H[\{\!\{\pi\}\!\}^{\mu}[\texttt{throw}\ \omega\ \texttt{in}\ \{\!\{\rho\}\!\}^{\mu}[\{\!\{t\}\!\}^{\lambda}_{e}]]]$. Par la règle de réduction du `throw`, on tire $\{\!\{s\}\!\} \to H[\{\!\{\rho\}\!\}^{\mu}[\{\!\{t\}\!\}^{\lambda}_{e}]]$. Or $\{\!\{s'\}\!\} = H[\{\!\{\rho\}\!\}^{\mu}[\{\!\{t\}\!\}^{\lambda}_{e}]]$. Donc $\{\!\{s\}\!\} \to \{\!\{s'\}\!\}$.

Autres cas. Les cas d'application de déconstructeurs représentent des passages sous le contexte, et la traduction est conservée à l'identique lors de la réduction (cf. le cas de l'application).

Les cas de valeurs avec déconstructeurs sur la pile représentent les δ-règles et la démonstration est analogue au cas de la β-réduction.

Le fait qu'on ne puisse pas avoir de bijection entre $\to$ et $\to_{\mathcal{M}}$ tient sa source dans le passage au contexte. En effet, la machine abstraite opère explicitement la déconstruction du contexte de tête lors de la réduction (et ce dernier constitue de fait la pile), alors que la sémantique opérationnelle le fait de manière implicite, par compatibilité du contexte.

Lemme 3.23 (Conservation de la terminaison) *Soit s un état de la machine abstraite. Si $s \not\to_{\mathcal{M}}$ alors $\{\!\{s\}\!\} \not\to$.*

Preuve On regarde le terme de la machine et on suppose $s \not\to_{\mathcal{M}}$.

Si $s = \langle x, e, \pi \rangle$**.** Nécessairement, $x \notin e$, sinon s se réduirait. Alors $\{\!\{s\}\!\} = H[\{\!\{\pi\}\!\}^{\mu}[x]] \not\to$.

Si $s = \langle C_i, e, \pi \rangle$**.** Nécessairement, $\pi \neq (\Lambda^{\iota} f_1 \dots f_k, e') :: \pi'$, sinon s se réduirait. Alors $\{\!\{s\}\!\} = H[\{\!\{\pi\}\!\}^{\mu}[C_i]]$, avec $\{\!\{\pi\}\!\}^{\mu}$ ne finissant pas par un déconstructeur de ι. D'après les propriétés des δ-règles, ce terme ne se réduit pas (en fait, il est mal typé).

Si $s = \langle K_i t, e, \pi \rangle$ **ou** $s = \langle (t_1, \dots, t_n), e, \pi \rangle$**.** Analogue au cas précédent.

Si $s = \langle \texttt{fun}\ x \mapsto t, e, \pi \rangle$**.** Nécessairement, $\pi \neq (u, e') :: \pi'$. Alors $\{\!\{s\}\!\} = H[\{\!\{\pi\}\!\}^{\mu}[\texttt{fun}\ x \mapsto \{\!\{t\}\!\}^{\lambda}_{e}]]$, avec $\{\!\{\pi\}\!\}^{\mu}$ ne finissant pas par une application. Ce terme ne se réduit pas dans CTML.

Si $s = \langle \texttt{throw}\ \alpha\ \texttt{in}\ t, e, \pi \rangle$**.** Nécessairement $\alpha \notin e$. Alors $\{\!\{s\}\!\} = H[\{\!\{\pi\}\!\}^{\mu}[\texttt{throw}\ \alpha\ \texttt{in}\ \{\!\{t\}\!\}^{\lambda}_{e}]]$ et ce terme ne se réduit pas.

Autres cas. Ils se réduisent toujours.

Théorème 3.24 (Correction de la machine abstraite) *Si t est tel que $H[t] \xrightarrow{*} H[v]$, alors le calcul de $s = \langle t, \emptyset, \varepsilon \rangle$ bloque sur un état s' tel que $\{\!\{s'\}\!\} = H[v]$.*

Preuve On montre aisément qu'on a $\{\!\{s\}\!\} = H[t]$. On montre le théorème par induction sur la longueur de la réduction.

1. Si $s \not\to_{\mathcal{M}}$, par le lemme 3.23, $H[t] \not\to$ et donc $\{\!\{s\}\!\} = \{\!\{s'\}\!\} = H[t] = H[v]$.
2. Si $s \to_{\mathcal{M}} s''$, par le lemme 3.21, $\{\!\{s\}\!\} \to \{\!\{s''\}\!\}$ ou $\{\!\{s\}\!\} = \{\!\{s''\}\!\}$. Or $\{\!\{s\}\!\} \xrightarrow{*} H[v]$. Par déterminisme de la réduction de terme, on a $\{\!\{s''\}\!\} \xrightarrow{*} H[v]$. Par hypothèse de récurrence, s'' se réduit et bloque sur un état s' tel que $\{\!\{s'\}\!\} = H[v]$, d'où le résultat.

3.3 Plongement dans le $\lambda\mu$-calcul

Pour se convaincre tout-à-fait du bien-fondé de la sémantique de CTML, on le plonge dans le $\lambda\mu$-calcul non typé, et on vérifie que le résultat est cohérent. C'est ce que nous assure le théorème 3.25.

On rappelle ici la définition inductive des termes t^λ du $\lambda\mu$-calcul, ainsi que les règles de réduction essentielles[14]. Cf. l'article fondateur de Parigot [6] pour de plus amples détails.

$$t^\lambda ::= x \mid t_1^\lambda t_2^\lambda \mid \lambda x.t^\lambda \mid \mu\alpha.t^\lambda \mid [\alpha]t^\lambda$$

$$\begin{array}{lcl} (\lambda x.t)u & \to & t\langle x \leftarrow u\rangle \\ (\mu\alpha.t)u & \to & \mu\alpha.t\langle [\alpha]w \leftarrow [\alpha]wu\rangle \\ [\alpha]\mu\beta.t & \to & t\langle \beta \leftarrow \alpha\rangle \end{array}$$

Définition 3.4 *On définit pour tout terme t de* CTML *un plongement $[\![t]\!]$ dans les termes du $\lambda\mu$-calcul.*

- $[\![x]\!] = x$;
- $[\![C_i]\!] = \lambda x_1.\ldots.\lambda x_i.\ldots.\lambda x_n.x_i$ *si* $\iota\{\bar\tau\} = E_1 \mid \ldots \mid C_i \mid \ldots \mid E_n$;
- $[\![K_i t]\!] = \lambda x_1.\ldots.\lambda x_i.\ldots.\lambda x_n.x_i[\![t]\!]$ *si* $\iota\{\bar\alpha\} = E_1 \mid \ldots \mid K_i\{\bar\tau\} \mid \ldots \mid E_n$;
- $[\![(t_1,\ldots,t_n)]\!] = \lambda f.f[\![t_1]\!]\ldots[\![t_n]\!]$;
- $[\![\texttt{fun}\ x \mapsto t]\!] = \lambda x.[\![t]\!]$;
- $[\![t_1 t_2]\!] = [\![t_1]\!][\![t_2]\!]$;
- $[\![\texttt{let}\ x = u\ \texttt{in}\ t]\!] = (\lambda x.[\![t]\!])[\![u]\!]$;
- $[\![\texttt{let rec}\ x = u\ \texttt{in}\ t]\!] = (\lambda x.[\![t]\!])(Y(\lambda x.[\![u]\!]))$ *où* $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$;
- $[\![\Pi^n\ t\ f]\!] = [\![t]\!][\![f]\!]$
- $[\![\Lambda^\iota\ t\ f_1 \ldots f_n]\!] = [\![t]\!][\![f_1]\!]\ldots[\![f_n]\!]$;
- $[\![\texttt{catch}\ \pi\ \texttt{in}\ t]\!] = \mu\pi.[\pi][\![t]\!]$;
- $[\![\texttt{throw}\ \pi\ \texttt{in}\ t]\!] = \mu\delta.[\pi][\![t]\!]$ *avec* $\delta \notin \mu fv(t)$;

Définition 3.5 *Pour tout terme t^λ du $\lambda\mu$-calcul, on note $t^\lambda \Downarrow_w v^\lambda$ si v^λ est une forme normale de t^λ pour la réduction en appel par nom.*

Théorème 3.25 *Soit t un terme de* CTML *Γ_0-typable, si $H[t] \xrightarrow{*} H[v]$, alors $[\![t]\!] \Downarrow_w u$ avec u isomorphe à $[\![H[v]]\!]$, c'est-à-dire égaux quitte à fusionner et α-convertir des μ en tête.*

[14] Il faut y ajouter les règles de compatibilité au contexte, qui conditionnent la stratégie de réduction. En appel par nom, si $t \to t'$ alors $tu \to t'u$ et $\mu\alpha.t \to \mu\alpha.t'$.

Conclusion

À partir d'un noyau ML en appel par nom, nous avons élaboré un langage proposant des primitives de contrôle inspirées du $\lambda\mu$-calcul. Afin de ne pas dérouter l'utilisateur habitué, CTML hérite d'une structure `try-with` comparable à celle des ML, quoique présentant des différences significatives en termes de sémantique. Ces différences sont dans l'ensemble bénéfiques tant pour la sécurité de l'exécution d'un programme que pour son efficacité. Comme nous l'avons montré, elles n'interfèrent pas avec les lignes de bonne conduite d'un ML que sont la correction conjointe du typage et de la sémantique. Elle constituent ainsi une extension complètement orthogonale de ML.

On ne peut que regretter que les exceptions statiques n'autorisent pas certaines opérations permises par les exceptions dynamiques habituelles des ML, qui sont d'une utilité pratique qu'il n'est pas permis de nier. Ces limitations sont le prix à payer pour les avantages apportés. Cependant, il semble que dans de nombreux cas, les exceptions statiques à l'œuvre en CTML soient supérieures à leur équivalent dynamique. Dans une vision œcuménique des langages ML, on pourrait imaginer un système mixte permettant l'emploi simultané d'exceptions statiques et dynamiques, à l'aide de la seule structure `try-with` associée à une discrimination lexicale. Nous déplorons le fait que des langages grand public comme OCaml ne connaissent pas de réels opérateurs de contrôle[15]. Cet état de fait semble lié à la complexité de compilation des opérateurs comme `callcc`.

Il serait possible d'étendre encore CTML avec d'autres constructions. Une extension que nous avions particulièrement en vue était l'ajout de structures impératives, ce qui est fréquent chez les ML. Cependant, le fait que CTML soit un langage paresseux ne fait pas bon ménage avec l'impératif, qui nécessite un séquencement défini des opérations. Il faudrait pour cela ajouter à CTML des structures qui forcent le calcul.

Enfin, l'interpréteur pourrait être amélioré, avec la mise en place d'une évaluation *call-by-need* ainsi que d'autres optimisations diverses liées à l'optimisation des opérations de contôle, assez coûteuses en soi.

Références

[1] Dubois, C., and Ménissier-Morain, V. Typage de ML : Spécification et preuve en Coq. In *Journées du GDR Programmation, 1998.*

[2] Laird, J. D. *A Semantic analysis of control.* PhD thesis, 1999.

[3] Laurent, O. Krivine's Abstract Machine and the $\lambda\mu$-calculus (an overview). 2003.

[4] Leroy, X. *The Objective Caml system (documentation and user's manual)*, 2008.

[5] Ong, C.-H. L. A semantic view of classical proofs : type-theoretic, categorical, and denotational characterizations. In *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*, pp. 230–241.

[6] Parigot, M. $\lambda\mu$-calculus : an Algorithmic Interpretation of Classical Natural Deduction. In *Logic Programming and Automated Reasoning : International Conference LPAR '92 Proceedings, St. Petersburg, Russia* (1992), A. Voronkov, Ed., Springer-Verlag, pp. 190–201.

[7] Rémy, D. Typage et Programmation. D.E.A. Sémantique, Preuves et Programmation, 1997.

[8] Rémy, D. Using, Understanding, and Unraveling the OCaml Language. In *Applied Semantics. Advanced Lectures. LNCS 2395.*, G. Barthe, Ed. Springer-Verlag, 2002, pp. 413–537.

[15] Citons quand même Xavier Leroy qui a publié un `callcc` naïf pour OCaml, hautement expérimental, inefficace au possible et réservé au *bytecode*.