Sémantique de C−− avec exceptions

Jean Goubault-Larrecq
ENS Cachan
goubault@lsv.ens-cachan.fr

28 novembre 2013

1 Introduction

Le but de ce document est de donner une sémantique à grands pas de C−− avec exceptions, le langage à compiler dans la phase 2 du projet de programmation 1. L'essentiel de ce document consiste à rappeler la sémantique de la phase 1, déjà donnée en section 1.6 du poly de « programmation 1 : sémantique, leçon 1 » : ce seront les sections 2 et 3. Les très rares changements seront marqués d'une note « nouveau : phase 2 » écrite dans la marge. La nouveauté, c'est la sémantique des exceptions donnée en section 4.

2 Considérations générales

Le language C−− travaille sur des valeurs non plus dans $\mathbb{Z}$, mais dans $\mathbb{Z}_k$, avec k un multiple de 8 strictement positif : $k = 32$ ou $k = 64$, typiquement. On définit $\mathbb{Z}_k$ comme étant l'intervalle des entiers compris entre -2^{k-1} et $2^{k-1} - 1$: on reconnaît le domaine des entiers *signés* sur k bits.

Le fait que C−− utilise aussi des pointeurs nécessite d'introduire une notion de *mémoire* μ, c'est-à-dire une fonction (partielle) qui à chaque adresse $a \in Addr$ (où $Addr$, l'ensemble des adresses, est l'ensemble des éléments des entiers relatifs multiples de 2^k) pour laquelle $\mu(a)$ est définie associe une valeur $\mu(a) \in \mathbb{Z}_k$. On notera $\mathrm{dom}\ \mu$ l'ensemble des adresses a telles que $\mu(a)$ est définie.

Un environnement ρ associe désormais à chaque variable x une *adresse* $a = \rho(x)$ où est stockée la valeur de x. En particulier, elle ne stocke plus la valeur de x directement. Pour l'obtenir, il faudra calculer $\mu(\rho(x))$.

Nous changeons un peu la notion d'environnement : désormais, un environnement sera aussi une fonction *partielle*. Les variables hors du domaine de l'environnement ρ seront celles qui n'ont pas été déclarées.

Un autre changement par rapport à IMP est que les commandes *et les* expressions peuvent avoir des effets de bord. Nous devrons donc inclure à la fois ρ et μ dans l'état.

Finalement, on décrira les expressions et commandes à évaluer dans la syntaxe des arbres de syntaxe abstraite Caml qui les décrivent. On omettra les « `location` » (qui servent d'aide aux messages d'erreur), et on identifiera donc les « `loc_expr` » et les « `expr` », les `loc_code` » et les « `code` ».

Un *programme* π est alors juste un objet du type `var_declaration list` du projet : une liste de :

- déclarations de variables `CDECL` x (pour un nom de variable x ; équivalent à la déclaration C `int` x) ;
- définitions de fonctions `CFUN`(f, ℓ, c) : ceci est équivalent à la déclaration C :

  `int` f `(`ℓ`) {`
  c
  `}`

  La liste ℓ est elle aussi de type `var_declaration list`, mais est *garantie* ne contenir que des déclarations de variables `CDECL` x_1, ..., `CDECL` x_n, et aucune définition de fonction (ce qui n'aurait aucun sens, au passage).
  On aura la même garantie dans les blocs `CBLOCK`$(\ell, [c_1; c_2; \cdots; c_n])$, où ℓ ne contiendra que des déclarations de variables. Ces derniers sont les équivalents du C : `{` ℓ c_1 ; c_2 ; ... ; c_n : `}`, où ℓ est une déclaration de variables locales (éventuellement vide, le bloc servant alors à coder la séquence).

Dans la suite,

on supposera un programme π fixé une fois pour toutes.

Ceci nous évitera de devoir écrire des jugements de la forme $\rho, \mu \vdash^{\pi} c \Rightarrow \rho', \mu'$: on s'économisera l'exposant π, et donc un peu de la lourdeur de la notation. C'est d'autant plus indiqué que cet indice ne serait utile que dans une règle : la règle (`CALL`) que nous verrons plus bas, et qui gère l'appel des fonctions.

On supposera de plus que π nous permet de définir un environnement ρ_{glob}, l'*environnement des variables globales*. Son domaine est l'ensemble des variables x telles que `CDECL` x est dans la liste π. On suppose que ρ_{glob} est *injective*, c'est-à-dire attribue à deux variables globales distinctes des adresses distinctes.

Cette sémantique ne gère pas la compilation séparée. Dans un souci de simplicité, on considérera que le programme est la concaténation de *tous* les fichiers sources le composant... y compris les sources des fonctions de la bibliothèque C standard comme `fprintf` ou `exit`, y compris les déclarations de variables globales comme `stderr`.

3 Phase 1 : tout C−−, sauf les exceptions

Expressions et l-values. On commence par définir des règles exprimant comme calculer l'adresses des expressions qui ont une adresse (les *l-values*, le « l » abrégeant « left », la gauche du symbole d'affectation).

La forme du jugement associé sera $\rho, \mu \vdash_{\&} e \Rightarrow V, \mu'$, où a est l'adresse souhaitée, et μ' la mémoire après l'exécution de l'évaluation de l'adresse de e. On définit donc :

$$\frac{}{\rho, \mu \vdash_{\&} \texttt{VAR}\ x \Rightarrow \rho(x), \mu}\ (\&\texttt{VAR})$$
$$\text{si } x \in \text{dom } \rho$$

$$\frac{\rho, \mu \vdash e_2 \Rightarrow n, \mu' \qquad \rho, \mu' \vdash e_1 \Rightarrow a, \mu''}{\rho, \mu \vdash_{\&} \texttt{OP2}(\texttt{S\_INDEX}, e_1, e_2) \Rightarrow a + n.2^k, \mu''}\ (\&\texttt{INDEX})$$
$$\text{si } n \in \mathbb{Z}_k, a \in Addr$$

Les deux prémisses de cette dernière règle affichent '$\vdash$' et non '$\vdash_{\&}$'. Ce n'est pas une erreur.

Notez aussi que l'on évalue *de droite à gauche* : c'est un choix de conception de C$--$ (que ne fait pas C), et qui est là pour vous simplifier la vie.

Les affectations. On peut affecter des variables ou des éléments de tableaux uniquement, en C$--$.

$$\frac{\rho, \mu \vdash e \Rightarrow V, \mu' \qquad \rho, \mu' \vdash_{\&} x \Rightarrow a, \mu''}{\rho, \mu \vdash \texttt{SET\_VAR}(x, e) \Rightarrow V, \mu''[a \mapsto V]}\ (\texttt{SET\_VAR})$$
$$\text{si } a \in \text{dom } \mu''$$

La condition de bord $a \in \text{dom } \mu''$ sert à assurer qu'une affectation se fasse dans une zone de mémoire préalablement allouée.

$$\frac{\rho, \mu \vdash e_2 \Rightarrow V, \mu' \qquad \rho, \mu' \vdash_{\&} \texttt{OP2}(\texttt{S\_INDEX}, x, e_1) \Rightarrow a, \mu''}{\rho, \mu \vdash \texttt{SET\_ARRAY}(x, e_1, e_2) \Rightarrow V, \mu''[a \mapsto V]}\ (\texttt{SET\_ARRAY})$$
$$\text{si } a \in \text{dom } \mu''$$

Variables, constantes. La règle (Rd) permet de lire la valeur d'une l-value. Ceci traite du cas des variables et des accès aux tableaux.

$$\frac{\rho, \mu \vdash_{\&} e \Rightarrow a, \mu'}{\rho, \mu \vdash e \Rightarrow \mu'(a), \mu'}\ (Rd)$$
$$\text{si } a \in \text{dom } \mu'$$

Pour $n \in \mathbb{Z}$, on notera $n \bmod 2^k$ l'unique élément de $\mathbb{Z}_k$ congru à n modulo 2^k.

$$\frac{}{\rho, \mu \vdash \texttt{CST}\ n \Rightarrow n \bmod 2^k, \mu}\ (\texttt{CST})$$

On a défini la mémoire μ comme un tableau de mots de 2^k bits. Nous pouvons aussi la voir comme un tableau d'octets, comme suit. Notons μ_{oct} la fonction partielle de Z_k vers l'ensemble des octets de domaine $\text{dom } \mu_{oct} = \{a \in Addr \mid (k/8)\lfloor a/(k/8)\rfloor \in \text{dom } \mu\}$, et telle que pour tout $a \in \text{dom } \mu$, $\mu(a) = \sum_{i=0}^{k/8-1} \mu_{oct}(a+i)256^i$.

Exercice 1 *Cette vue de la mémoire est-elle petit-boutiste ? grand-boutiste ?*

Soit s une chaîne de caractères, disons formée des caractères s_0, s_1, …, s_{n-1} dans cet ordre. Nous dirons que la mémoire μ *contient* s *à l'adresse* a si et seulement si les adresses a, $a+1$, …, $a+n-1$, $a+n$ sont toutes dans $\mathrm{dom}\ \mu_{oct}$, $\mu_{oct}(a) = s_0$, $\mu_{oct}(a+1) = s_1$, …, $\mu_{oct}(a+n-1) = s_{a+n-1}$, et $\mu_{oct}(a+n) = 0$.

$$\frac{}{\rho, \mu \vdash \texttt{STRING}\ s \Rightarrow a, \mu} \ (\texttt{STRING})$$
$$\text{si } \mu \text{ contient } s \text{ à l'adresse } a$$

Cette règle peut paraître un peu curieuse : comment assure-t-on qu'il y a une adresse où la chaîne s serait stockée ? S'il y en a plusieurs, laquelle retourne-t-on ?

La réponse est qu'on laisse toute liberté au compilateur pour trouver une telle adresse. S'il n'y en a pas, la sémantique du programme sera indéfinie. En pratique, on s'arrangera pour préallouer la chaîne de caractères s à une adresse a, et le compilateur produire du code assembleur qui se contente de fournir cette adresse.

Il est hors de question d'*allouer* l'adresse a au moment de l'évaluation de `STRING` s : ce n'est pas ce que nous souhaitons que ce genre d'expression fasse. On veut juste retourner l'adresse d'une zone mémoire où la chaîne s est *déjà* stockée.

Les opérations monadiques.

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu'}{\rho, \mu \vdash \texttt{OP1}(\texttt{M\_MINUS}, e) \Rightarrow (-n) \bmod 2^k, \mu'} \ (\texttt{MINUS})$$

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu'}{\rho, \mu \vdash \texttt{OP1}(\texttt{M\_NOT}, e) \Rightarrow (-1-n) \bmod 2^k, \mu'} \ (\texttt{NOT})$$

Exercice 2 *À quoi peut-on voir que les entiers sont codés en complément à deux dans cette sémantique ?*

Voici la sémantique de la post-incrémentation ; en syntaxe concrète, `x++` ou `a[i]++`, par exemple.

$$\frac{\rho, \mu \vdash_{\&} e \Rightarrow a, \mu'}{\rho, \mu \vdash \texttt{OP1}(\texttt{M\_POST\_INC}, e) \Rightarrow n, \mu'[a \mapsto (n+1) \bmod 2^k]} \ (\texttt{POST\_INC})$$
$$\text{si } a \in \mathrm{dom}\ \mu', n = \mu'(a)$$

De même, la post-décrémentation, `x--` ou `a[i]--` :

$$\frac{\rho, \mu \vdash_{\&} e \Rightarrow a, \mu'}{\rho, \mu \vdash \texttt{OP1}(\texttt{M\_POST\_DEC}, e) \Rightarrow n, \mu'[a \mapsto (n-1) \bmod 2^k]} \ (\texttt{POST\_DEC})$$
$$\text{si } a \in \mathrm{dom}\ \mu', n = \mu'(a)$$

De façon symétrique, la pré-incrémentation, `++x` ou `++a[i]` :

$$\frac{\rho, \mu \vdash_{\&} e \Rightarrow a, \mu'}{\rho, \mu \vdash \texttt{OP1}(\texttt{M\_PRE\_INC}, e) \Rightarrow (n+1) \bmod 2^k, \mu'[a \mapsto (n+1) \bmod 2^k]} \ (\texttt{PRE\_INC}) \quad \text{si } a \in \operatorname{dom} \mu', n = \mu'(a)$$

et la pré-décrémentation, `--x` ou `--a[i]` :

$$\frac{\rho, \mu \vdash_{\&} e \Rightarrow a, \mu'}{\rho, \mu \vdash \texttt{OP1}(\texttt{M\_PRE\_DEC}, e) \Rightarrow (n-1) \bmod 2^k, \mu'[a \mapsto (n-1) \bmod 2^k]} \ (\texttt{PRE\_DEC}) \quad \text{si } a \in \operatorname{dom} \mu', n = \mu'(a)$$

Les (autres) opérations binaires. Nous avons déjà traité de l'opération `S_INDEX`, au travers des règles ($\&\texttt{INDEX}$) et (Rd). Regardons les autres.

On évalue toujours de droite à gauche en C−−.

$$\frac{\rho, \mu \vdash e_2 \Rightarrow n_2, \mu' \qquad \rho, \mu' \vdash e_1 \Rightarrow n_1, \mu''}{\rho, \mu \vdash \texttt{OP2}(\texttt{S\_ADD}, e_1, e_2) \Rightarrow (n_1 + n_2) \bmod 2^k, \mu''} \ (\texttt{ADD})$$

$$\frac{\rho, \mu \vdash e_2 \Rightarrow n_2, \mu' \qquad \rho, \mu' \vdash e_1 \Rightarrow n_1, \mu''}{\rho, \mu \vdash \texttt{OP2}(\texttt{S\_SUB}, e_1, e_2) \Rightarrow (n_1 - n_2) \bmod 2^k, \mu''} \ (\texttt{SUB})$$

$$\frac{\rho, \mu \vdash e_2 \Rightarrow n_2, \mu' \qquad \rho, \mu' \vdash e_1 \Rightarrow n_1, \mu''}{\rho, \mu \vdash \texttt{OP2}(\texttt{S\_MUL}, e_1, e_2) \Rightarrow (n_1 \times n_2) \bmod 2^k, \mu''} \ (\texttt{MUL})$$

Pour la division, définissons $\lfloor x \rceil$ pour chaque nombre réel x comme étant l'entier de même signe que x dont la valeur absolue est la partie entière de la valeur absolue de x. Donc $\lfloor 2{,}3 \rceil = 2$, mais $\lfloor -2{,}3 \rceil = -2$.

$$\frac{\rho, \mu \vdash e_2 \Rightarrow n_2, \mu' \qquad \rho, \mu' \vdash e_1 \Rightarrow n_1, \mu''}{\rho, \mu \vdash \texttt{OP2}(\texttt{S\_DIV}, e_1, e_2) \Rightarrow \lfloor n_1/n_2 \rceil \bmod 2^k, \mu''} \ (\texttt{DIV}) \quad \text{si } n_2 \neq 0$$

$$\frac{\rho, \mu \vdash e_2 \Rightarrow n_2, \mu' \qquad \rho, \mu' \vdash e_1 \Rightarrow n_1, \mu''}{\rho, \mu \vdash \texttt{OP2}(\texttt{S\_MOD}, e_1, e_2) \Rightarrow (n_1 - n_2 \times \lfloor n_1/n_2 \rceil) \bmod 2^k, \mu''} \ (\texttt{MOD}) \quad \text{si } n_2 \neq 0$$

Exercice 3 *Soit $n_2 \neq 0$. Montrer que $n_1 - n_2 \times \lfloor n_1/n_2 \rceil$ est le reste de la division de $|n_1|$ par $|n_2|$ si $n_1 \geq 0$, et est l'opposé de ce reste si $n_1 \leq 0$.*

Les bizarreries dans la sémantique de la division et du reste sont, bien sûr, faites pour correspondre à la sémantique des mêmes opérations en assembleur x86 (et de la plupart des autres processeurs).

Appels de fonctions Il y a beaucoup à expliquer ici.

D'abord, encore une fois, on évalue les arguments de droite à gauche.

Ensuite, on a besoin d'allouer de la mémoire, pour allouer la place (sur la pile dans votre compilateur) pour les paramètres de la fonction f. Pour ceci, on suppose une fonction $alloc$ qui a chaque mémoire μ associe, soit une adresse $a \in Addr$ qui n'est pas dans $\mathrm{dom}\ \mu$ (qui n'est pas encore allouée), soit le symbole spécial $\bot$, supposé hors de $Addr$ (ceci représente typiquement l'épuisement de l'espace mémoire disponible).

$$\frac{\begin{array}{rcl}\rho, \mu & \vdash & e_k \Rightarrow n_k, \mu_k \\ \rho, \mu_k[a_k \mapsto n_k] & \vdash & e_{k-1} \Rightarrow n_{k-1}, \mu_{k-1} \\ & \vdots & \\ \rho, \mu_2[a_2 \mapsto n_2] & \vdash & e_1 \Rightarrow n_1, \mu_1 \\ \rho_{\mathrm{glob}}[x_1 \mapsto a_1, \cdots, x_k \mapsto a_k], \mu_1[a_1 \mapsto n_1] & \vdash & c \Rightarrow \texttt{ret}\ n, \mu'\end{array}}{\begin{array}{r}\rho, \mu \vdash \texttt{CALL}(f, [e_1; \cdots; e_k]) \Rightarrow n, \mu' \\ \text{s'il existe une unique entrée } \texttt{CFUN}(g, [x_1; \cdots; x_{k'}], c) \\ \text{avec } g = f \text{ dans } \pi \\ \text{et si } k' = k, \\ \text{et si les } x_i \text{ sont disjoints } (1 \leq i \leq k), \\ a_k = alloc(\mu_k) \neq \bot, \\ \cdots, \\ a_2 = alloc(\mu_2) \neq \bot, \\ a_1 = alloc(\mu_1) \neq \bot\end{array}} (\texttt{CALL})$$

On notera une nouvelle forme de jugement (la dernière prémisse), où ce que l'on évalue n'est plus une expression mais une commande (c), et l'on retourne non pas une valeur et une mémoire, mais :

- un *paquet de retour* `ret` n ; ceci sera la « valeur » de l'instruction `return e` lorsque la valeur de `e` est l'entier n ;
- et une mémoire μ'.

La règle (`CALL`) ne gère pas le cas où la fonction appelée a lancé une exception qu'elle n'a pas rattrapée par `catch`. Ceci sera traité par la règle (`CALL`Exc) en section 4. Nouveau : phase 2

Commandes Commençons par l'instruction `return` avec argument (par exemple, `return x+1`) :

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu'}{\rho, \mu \vdash \texttt{CRETURN}\ (\texttt{Some}\ e) \Rightarrow \texttt{ret}\ n, \mu'}\ (\texttt{RETURN})$$

Ensuite, l'instruction `return` sans argument :

$$\frac{}{\rho, \mu \vdash \texttt{CRETURN}\ (\texttt{None}) \Rightarrow \texttt{ret}\ n, \mu'}\ (\texttt{RETURNO})$$

où $n \in \mathbb{Z}_k$

qui donc retourne un entier machine *arbitraire*. (La sémantique est donc non déterministe, en particulier.) En C, la question ne se pose pas : le système de typage de C garantit qu'on ne

peut écrire `return` sans argument que si la fonction courante a le type de retour `void`, ce qui garantit que quelque valeur que nous retournions, celle-ci ne sera de toute façon jamais utilisée. En C−−, il n'y a pas de typage... et nous décidons de retourner une valeur quelconque. C'est (de nouveau) fait pour vous faciliter la vie : si, comme recommandé dans le projet, vous décidez de placer la valeur retournée dans `%eax` (resp., `%rax`), ceci vous permet de ne pas vous préoccuper du contenu de ce registre du tout lors des `return` sans argument.

Pour les autres commandes, le jugement typique aura la forme $\rho, \mu \vdash c \Rightarrow \mu'$, où μ' est la mémoire obtenue lorsque l'exécution de c se termine normalement (sans passer par un `return`). Par exemple, pour les expressions vues comme commandes :

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu'}{\rho, \mu \vdash \texttt{CEXPR}\ e \Rightarrow \mu'}\ (\texttt{CEXPR})$$

On se retrouve donc avec deux sortes de jugements pour une commande c : $\rho, \mu \vdash c \Rightarrow \mu'$ si c se termine normalement, et $\rho, \mu \vdash c \Rightarrow \texttt{ret}\ n, \mu'$ si c effectue un `return` d'une valeur n.

Pour une séquence $c_1; c_2$, nous devrons écrire que : soit c_1 effectue un `return` et l'on n'évalue pas c_2, soit c_1 se termine normalement, et on évalue c_2. Nous allons factoriser la sémantique des séquences dans une série de règles établissant des jugements de la forme $\rho, \mu \vdash L \Rightarrow \mu'$ ou $\rho, \mu \vdash L \Rightarrow \texttt{ret}\ n, \mu'$, où L est une *liste* finie de commandes, à exécuter en séquence. On note la liste vide [] comme en Caml, et celle obtenue en ajoutant c en tête de L sera notée $c :: L$. On notera res pour un ***résultat***, c'est-à-dire :

- une mémoire μ' (si la liste ou la commande se termine normalement),
- ou bien un couple $\texttt{ret}\ n, \mu'$ formé d'un paquet de retour et d'une mémoire (si la liste ou la commande effectue un `return`) ;
- ou bien d'un paquet d'exception (voir la section 4). Nouveau : phase 2

$$\frac{}{\rho, \mu \vdash [] \Rightarrow \mu}\ (Seq0)$$

$$\frac{\rho, \mu \vdash c \Rightarrow \mu' \qquad \rho, \mu' \vdash L \Rightarrow res}{\rho, \mu \vdash c :: L \Rightarrow res}\ (Seq1)$$

$$\frac{\rho, \mu \vdash c \Rightarrow \texttt{ret}\ n, \mu'}{\rho, \mu \vdash c :: L \Rightarrow \texttt{ret}\ n, \mu'}\ (SeqRet)$$

En phase 2, au vu de l'ajout des paquets d'exception , il nous faudra une règle supplémentaire, la règle (SeqExc), que nous verrons en section 4. Nouveau : phase 2

On peut maintenant donner la sémantique des blocs, qui allouent de la place pour des va-

riables locales, puis effectuent une séquence de commandes.

$$\frac{\rho[x_1 \mapsto a_1, \cdots, x_k \mapsto a_k], \mu_k \vdash L \Rightarrow res}{\rho, \mu \vdash \texttt{CBLOCK}([x_1; \cdots; x_k], L) \Rightarrow res}\ (\texttt{CBLOCK})$$
$$\text{où } a_1 = alloc(\mu) \neq \bot,$$
$$n_1 \in \mathbb{Z}_k, \mu_1 = \mu[a_1 \mapsto n_1],$$
$$a_2 = alloc(\mu_1) \neq \bot,$$
$$n_2 \in \mathbb{Z}, \mu_2 = \mu_1[a_2 \mapsto n_2],$$
$$\ldots, a_k = alloc(\mu_{k-1}) \neq \bot,$$
$$n_k \in \mathbb{Z}, \mu_k = \mu_{k-1}[a_k \mapsto n_k]$$

Ci-dessus, les entiers n_1, …, n_k ne sont soumis à aucune contrainte. Ils sont donc *quelconques* : comme en C, les variables nouvellement déclarées ne sont pas initialisées. (Il en est de même des variables globales, si vous regardez bien.)

Il ne reste plus qu'à traiter de la conditionnelle :

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu' \qquad \rho, \mu' \vdash c_1 \Rightarrow res}{\rho, \mu \vdash \texttt{CIF}(e, c_1, c_2) \Rightarrow res}\ (\texttt{CIF}_1)$$
$$\text{si } n \neq 0$$

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu' \qquad \rho, \mu' \vdash c_2 \Rightarrow res}{\rho, \mu \vdash \texttt{CIF}(e, c_1, c_2) \Rightarrow res}\ (\texttt{CIF}_2)$$
$$\text{si } n = 0$$

et des boucles `while` :

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu' \quad \rho, \mu' \vdash [c; \texttt{CWHILE}(e, c)] \Rightarrow res}{\rho, \mu \vdash \texttt{CWHILE}(e, c) \Rightarrow res}\ (\texttt{CWHILE})$$
$$\text{si } n \neq 0$$

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu'}{\rho, \mu \vdash \texttt{CWHILE}(e, c) \Rightarrow \mu'}\ (\texttt{CWHILE}_{fin})$$
$$\text{si } n = 0$$

4 Phase 2 : les exceptions

Par rapport à la phase 1, le C−− de la phase 2 contient deux nouvelles constructions d'arbre de syntaxe abstraite :

- $\texttt{CTHROW}(E, e)$, où E est une chaîne de caractères appelée *nom d'exception* et e est une expression, correspondant à la syntaxe concrète `throw` $E(e)$;
- $\texttt{CTRY}(c, [(E_1, x_1, c_1); (E_2, x_2, c_2); \cdots; (E_n, x_n, c_n)], F)$, où c est une commande, chaque E_i est un nom d'exception, chaque x_i est un nom de variable (une chaîne de caractères), chaque c_i est une commande, et F vaut soit `None` soit `Some` c'. Chaque triplet (E_i, x_i, c_i) est appelé un *handler* d'exception.
  La syntaxe concrète est

```
try {
    c
}
catch (E1 x1) { c1 }
catch (E2 x2) { c2 }
...
catch (En xn) { cn }
```

avec la ligne supplémentaire :
`finally { c' }`
si $F =$ `Some` c'. L'idée est d'exécuter c. Si c lance une exception E_i avec valeur V_i, on doit ensuite exécuter c_i avec V_i stocké dans la variable x_i. Que c retourne normalement, en lançant une exception, ou en effectuant un `return`, le code c' de la clause `finally` doit être exécuté à la fin (au moins dans les cas les plus courants... il est impossible de le garantir dans tous les cas).

On notera que si C−−, comme C, est un langage à liaison lexicale, les portées des clauses `catch` sont dynamiques : il est impossible de savoir vers quelle clause `catch` l'exécution va se brancher lorsqu'un `throw` est exécuté, à la lecture seule du code.

Pour définir la sémantique de ces nouvelles constructions, nous allons introduire, en plus des valeurs V et des paquets de retour `ret` V, des *paquets d'exception* de la forme $\texttt{throw}(E, n)$, où E est un nom d'exception et n est une valeur.

Lancement d'exception.

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu'}{\rho, \mu \vdash \texttt{CTHROW}(E, e) \Rightarrow \texttt{throw}(E, n), \mu'} \ (\texttt{CTHROW})$$

Séquence : règle supplémentaire. Comme les paquets de retour, les paquets d'exception interrompent l'exécution séquentielle :

$$\frac{\rho, \mu \vdash c \Rightarrow \texttt{throw}(E, n), \mu'}{\rho, \mu \vdash c :: L \Rightarrow \texttt{throw}(E, n), \mu'} \ (SeqExc)$$

Appel de fonctions : règle supplémentaire. Si une fonction appelée lance une exception qu'elle n'a pas elle-même rattrapée, l'exception est propagée à l'appelant :

$$\frac{\begin{array}{rcl}\rho, \mu & \vdash & e_k \Rightarrow n_k, \mu_k \\ \rho, \mu_k[a_k \mapsto n_k] & \vdash & e_{k-1} \Rightarrow n_{k-1}, \mu_{k-1} \\ & \vdots & \\ \rho, \mu_2[a_2 \mapsto n_2] & \vdash & e_1 \Rightarrow n_1, \mu_1 \\ \rho_{\text{glob}}[x_1 \mapsto a_1, \cdots, x_k \mapsto a_k], \mu_1[a_1 \mapsto n_1] & \vdash & c \Rightarrow \texttt{throw}(E, n), \mu'\end{array}}{\rho, \mu \vdash \texttt{CALL}(f, [e_1; \cdots; e_k]) \Rightarrow \texttt{throw}(E, n), \mu'} \; (\texttt{CALL}Exc)$$

s'il existe une unique entrée $\texttt{CFUN}(g, [x_1; \cdots; x_{k'}], c)$
avec $g = f$ dans π
et si $k' = k$,
et si les x_i sont disjoints $(1 \leq i \leq k)$,
$a_k = alloc(\mu_k) \neq \bot$,
$\cdots$,
$a_2 = alloc(\mu_2) \neq \bot$,
$a_1 = alloc(\mu_1) \neq \bot$

(C'est le moment de jouer aux jeux des 7 différences avec la règle (CALL). Si l'on exclut le nom de la règle, il n'y en a que deux.)

CTRY : conventions. Dans la suite, notons HL les listes de handlers d'exception. Notre but est de définir la sémantique de la construction $\texttt{CTRY}(c, HL, F)$.

Par souci de simplicité, nous identifierons $F = \texttt{None}$ avec $\texttt{Some}(\texttt{CBLOCK}([], []))$. Ceci nous permettra de ne *jamais* traiter du cas $F = \texttt{None}$. Nous traiterons donc uniquement de la construction $\texttt{CTRY}(c, HL, \texttt{Some}\ c')$.

Je n'expliquerai pratiquement rien de ces règles : c'est à vous de les lire, et de les relire. Il y a une multitude de façons « intuitives » d'implémenter la gestion des CTRY, toutes ne réaliseront pas la sémantique décrite ci-dessous.

CTRY : pas d'exception lancée.

$$\frac{\rho, \mu \vdash c \Rightarrow \mu' \quad \rho, \mu' \vdash c' \Rightarrow res}{\rho, \mu \vdash \texttt{CTRY}(c, HL, \texttt{Some}\ c') \Rightarrow res} \; (\texttt{CTRY})$$

CTRY : return dans le corps.

$$\frac{\rho, \mu \vdash c \Rightarrow \texttt{ret}\ n, \mu' \quad \rho, \mu' \vdash c' \Rightarrow \mu''}{\rho, \mu \vdash \texttt{CTRY}(c, HL, \texttt{Some}\ c') \Rightarrow \texttt{ret}\ n, \mu''} \; (\texttt{CTRY}Ret)$$

On rappelle que res désigne un résultat (mémoire, paquet de retour, ou paquet d'exception), alors que μ'' est contraint à être une mémoire. On notera paq pour désigner un *paquet*, paquet de retour

ou paquet d'exception — ceci nous permet d'écrire une règle au lieu de deux :

$$\frac{\rho, \mu \vdash c \Rightarrow \texttt{ret}\ n, \mu' \quad \rho, \mu' \vdash c' \Rightarrow paq}{\rho, \mu \vdash \texttt{CTRY}(c, HL, \texttt{Some}\ c') \Rightarrow paq}\ (\texttt{CTRY}RetPaq)$$

`CTRY` : lancement d'exception dans le corps.

$$\frac{\rho, \mu \vdash c \Rightarrow \texttt{throw}(E, n), \mu' \quad \rho, \mu', \texttt{throw}(E, n) \vdash HL\ \texttt{finally}\ c' \Rightarrow res}{\rho, \mu \vdash \texttt{CTRY}(c, HL, \texttt{Some}\ c') \Rightarrow res}\ (\texttt{CTRY}Exc)$$

On notera ici l'utilisation en second prémisse d'une nouvelle forme de jugement, exprimant que dans un environnement ρ, partant d'une mémoire μ', on cherche à trouver un handler d'exception (E_i, x_i, c_i) dans la liste HL et le traiter, ainsi que d'exécuter le code c', ceci se terminant sur le résultat res. Les règles de dérivation de ces nouveaux jugements sont données dans la suite.

`CTRY` : lancement d'exception dans le corps, pas de handler correspondant.

$$\frac{\rho, \mu' \vdash c' \Rightarrow \mu''}{\rho, \mu', \texttt{throw}(E, n) \vdash [\,]\ \texttt{finally}\ c' \Rightarrow \texttt{throw}(E, n), \mu''}\ (\texttt{catch}_{fin})$$

$$\frac{\rho, \mu' \vdash c' \Rightarrow paq}{\rho, \mu', \texttt{throw}(E, n) \vdash [\,]\ \texttt{finally}\ c' \Rightarrow paq}\ (\texttt{catch}_{fin}Paq)$$

`CTRY` : lancement d'exception dans le corps, recherche de handler correspondant.

$$\frac{\rho, \mu', \texttt{throw}(E, n) \vdash HL\ \texttt{finally}\ c' \Rightarrow res}{\rho, \mu', \texttt{throw}(E, n) \vdash (E_1, x_1, c_1) :: HL\ \texttt{finally}\ c' \Rightarrow res}\ (\texttt{catch}Non)$$
si $E \neq E_1$

$$\frac{\rho[x_1 \mapsto a_1], \mu'[a_1 \mapsto n_1] \vdash [c_1; c'] \Rightarrow res}{\rho, \mu', \texttt{throw}(E, n) \vdash (E_1, x_1, c_1) :: HL\ \texttt{finally}\ c' \Rightarrow res}\ (\texttt{catch}Oui)$$
si $E = E_1$
où $a_1 = alloc(\mu') \neq \bot$

Dans la règle $(\texttt{catch}Oui)$, la notation $[c_1; c']$, inspirée de Caml, désigne la liste $c_1 :: c' :: []$.