Langages de programmation: sémantique et typage
Examen partiel

13 décembre 2005

1 Introduction

On s'intéresse au typage de la construction callcc. Cette construction, très populaire dans le langage non typé Scheme, a été introduite dans le langage fortement typé Standard ML aux alentours de 1989. La règle de typage associée à callcc semblait simple et sûre : on découvrit pourtant, en 1991, que son interaction avec le polymorphisme de ML compromettait la sûreté du système de types. Nous étudions ici le contre-exemple découvert en 1991, expliquons pourquoi le système de types initial n'était pas sûr, et démontrons la sûreté du système après correction.

Indications Il n'est pas nécessaire d'avoir répondu à une question pour pouvoir traiter celles qui suivent. Les questions 1 et 2 font appel au calcul plus qu'au raisonnement ; les questions 3, 4 et 5 font appel au raisonnement plus qu'au calcul. Toutes demandent un certain temps, à l'exception de la question 3 qui peut être traitée relativement rapidement.

2 Sémantique

Syntaxe Le langage considéré est le λ-calcul doté d'une construction let et des constructions primitives callcc et abort. La syntaxe du langage est donc la suivante :

$$\begin{array}{rcl} e & ::= & v \mid v\ v \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{abort}\ e \\ v & ::= & x \mid \lambda x.e \mid \mathsf{callcc} \end{array}$$

L'opération abort n'est pas offerte au programmeur, mais est utile pour formaliser la sémantique opérationnelle. L'application est restreinte aux valeurs (on écrit $v\ v$ au lieu de $e\ e$) pour simplifier certaines preuves ; aucune expressivité n'est perdue de ce fait.

Évaluation La sémantique opérationnelle à petits pas est la suivante :

$$\begin{array}{rcll} E[(\lambda x.e)\ v] & \rightarrow & E[[x \mapsto v]e] & (1) \\ E[\mathsf{let}\ x = v\ \mathsf{in}\ e] & \rightarrow & E[[x \mapsto v]e] & (2) \\ E[\mathsf{callcc}\ v] & \rightarrow & E[v\ (\lambda x.\mathsf{abort}\ E[x])] & (3) \\ E[\mathsf{abort}\ e] & \rightarrow & e & (4) \end{array}$$

La méta-variable E représente un contexte d'évaluation, dont la syntaxe est donnée plus bas. La règle (4) indique que abort e abandonne l'exécution du calcul en cours – cela est reflété par le fait que le contexte d'évaluation E est perdu – et le remplace par l'évaluation de e. Le résultat final du programme sera donc le résultat de e. Par exemple, l'expression $1 + \mathsf{abort}\ 3$ a pour résultat 3. La règle (3) indique que callcc *réifie* le contexte d'évaluation courant : E est transformé en une fonction $\lambda x.\mathsf{abort}\ E[x]$, appelée *continuation*. Cette fonction est mise à la disposition de la fonction v. Si, au cours d'un calcul, v fait appel à la continuation, alors le contexte d'évaluation courant sera supprimé et le contexte E sera réinstallé à sa place. Ce mécanisme puissant et complexe a de nombreuses applications, qu'il m'est impossible d'expliquer ici.

La syntaxe des contextes d'évaluation est donnée par

$$E ::= [\,] \mid \mathsf{let}\ x = E\ \mathsf{in}\ e$$

On notera que $e \to e'$ n'implique pas $E[e] \to E[e']$. En effet, les constructions callcc et abort réifient ou détruisent *l'intégralité* du contexte englobant.

Pour les besoins des questions 1, 2 et 3, on ajoute au langage quelques constructions primitives : constantes entières et booléennes, addition entière, paires et projections. Les règles de réduction et de typage associées à ces constructions sont celles traitées en cours.

Question 1 *Expliciter la séquence de réduction issue du terme suivant :*

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{callcc}\ (\lambda k.(\lambda x.x, \lambda f.k\ (f, \lambda d.5)))\ \mathsf{in} \\ (\lambda z.\pi_2\ x\ (\lambda x.x+1))\ (\pi_1\ x\ \mathsf{true})\end{array}$$

et vérifier qu'elle conduit à un terme bloqué. Cette séquence de réduction étant relativement longue, on n'hésitera pas à abréger certains sous-termes à l'aide de points de suspension, pourvu qu'ils soient reconnaissables à leur préfixe ($\lambda k.\ldots$, $\lambda z.\ldots$, etc.). ◇

Solution. Voici la séquence de réductions :

$$\begin{array}{lll}
 & \mathsf{let}\ x = \mathsf{callcc}\ (\lambda k.(\lambda x.x, \lambda f.k\ (f, \lambda d.5)))\ \mathsf{in} & \\
 & (\lambda z.\pi_2\ x\ (\lambda x.x+1))\ (\pi_1\ x\ \mathsf{true}) & \\
\to & \mathsf{let}\ x = (\lambda k.\ldots)\ (\lambda x.\mathsf{abort}\ E[x])\ \mathsf{in} & \\
 & (\lambda z.\pi_2\ x\ (\lambda x.x+1))\ (\pi_1\ x\ \mathsf{true}) & \text{où } E = \mathsf{let}\ x = [\,]\ \mathsf{in}\ (\lambda z.\ldots)\ (\pi_1\ \ldots) \\
\to & \mathsf{let}\ x = (\lambda x.x, \lambda f.(\lambda x.\mathsf{abort}\ E[x])\ (f, \lambda d.5))\ \mathsf{in} & \\
 & (\lambda z.\pi_2\ x\ (\lambda x.x+1))\ (\pi_1\ x\ \mathsf{true}) & \\
\to^2 & (\lambda z.\pi_2\ (\lambda x.x, \lambda f.\ldots)\ (\lambda x.x+1))\ ((\lambda x.x)\ \mathsf{true}) & \\
\to & (\lambda z.\pi_2\ (\lambda x.x, \lambda f.\ldots)\ (\lambda x.x+1))\ \mathsf{true} & \\
\to & \pi_2\ (\lambda x.x, \lambda f.\ldots)\ (\lambda x.x+1) & \\
\to & (\lambda f.\ldots)\ (\lambda x.x+1) & \\
\to & (\lambda x.\mathsf{abort}\ E[x])\ (\lambda x.x+1, \lambda d.5) & \\
\to & \mathsf{abort}\ E[(\lambda x.x+1, \lambda d.5)] & \\
\to & E[(\lambda x.x+1, \lambda d.5)] & \\
= & \mathsf{let}\ x = (\lambda x.x+1, \lambda d.5)\ \mathsf{in}\ (\lambda z.\pi_2\ x\ \ldots)\ (\pi_1\ x\ \mathsf{true}) & \\
\to^2 & (\lambda z.\pi_2\ (\lambda x.x+1, \lambda d.5)\ \ldots)\ ((\lambda x.x+1)\ \mathsf{true}) & \\
\to & (\lambda z.\ldots)\ (\mathsf{true}+1) &
\end{array}$$

Le résultat obtenu est un terme bloqué. □

3 Typage

La syntaxe des types est standard : elle comprend au minimum les variables de types, les types flèche, un type unit, et peut également comprendre le type int des entiers, le type bool des booléens, les types produits, etc. Le type unit est utilisé ici dans la règle de typage de abort, mais tout autre type *ground* (sans variables de types) pourrait jouer ce rôle.

Typage de callcc et abort Les règles de typage sont celles du noyau de ML (c'est-à-dire le λ-calcul avec let), auxquelles on ajoute les deux règles suivantes :

$$\textsc{Callcc}\quad \Gamma \vdash \mathsf{callcc} : ((\tau_1 \to \tau_2) \to \tau_1) \to \tau_1 \qquad\qquad \textsc{Abort}\quad \frac{\Gamma \vdash e : \mathsf{unit}}{\Gamma \vdash \mathsf{abort}\ e : \tau}$$

On peut tenter d'expliquer la première règle comme suit. (Mais il n'est pas strictement nécessaire de lire cette explication pour répondre aux questions qui suivent ! Je digresse.) Le résultat de l'expression callcc $(\lambda k.e)$

est soit le résultat fourni par e (dans le cas où e n'applique pas la continuation k), soit l'argument fourni par e à k (dans le cas où e applique k). Si l'expression $\mathsf{callcc}\ (\lambda k.e)$ doit avoir le type τ_1, il faut donc exiger que e ait le type τ_1 et que l'argument fourni à k ait également le type τ_1. Cela explique les trois occurrences de τ_1 dans le type de callcc. Quant à τ_2, il représente le type du résultat produit par k. Aucune contrainte ne porte sur ce type, car une continuation k ne produit en fait aucun résultat : parce que son corps est constitué d'une application de abort, elle ne rend jamais la main à son appelant. Une autre façon de lire cette règle de typage consiste à l'examiner à la lumière de l'isomorphisme de Curry-Howard. Le type de callcc, lu comme une proposition logique, dit : « donne-moi une preuve de τ_1 sous l'hypothèse que τ_1 implique τ_2 ; je te donnerai une preuve de τ_1 sans cette hypothèse. » Il n'est pas difficile de se convaincre informellement que cet énoncé a un sens et que sa démonstration la plus naturelle correspond au comportement opérationnel de callcc.

La seconde règle de typage indique que l'expression abort e admet un type τ quelconque, ce qui reflète, encore une fois, le fait que cette expression ne renvoie aucun résultat au contexte qui l'entoure – justement, elle détruit ce contexte. L'argument e devient le résultat du programme complet. Il doit avoir le type unit, car on s'intéresse ici uniquement à des programmes dont le résultat final a le type unit. (On pourrait relaxer cette hypothèse, mais cela serait sans grand intérêt.)

Traitement du polymorphisme Pour simplifier les choses, on considérera une présentation du système de types où les règles de généralisation et d'instantiation n'existent pas séparément.

Nous nous intéressons d'une part à un système de types « permissif » dans lequel la construction let permet toujours la généralisation :

$$\textsc{Let}\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \bar{\alpha} \# \mathrm{ftv}(\tau_1) \qquad \Gamma; x : \forall \bar{\alpha}.\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

Nous nous intéressons d'autre part au système « restreint » obtenu en appliquant la restriction du polymorphisme aux valeurs (*value restriction*). La règle Let est alors remplacée par les deux règles suivantes :

$$\textsc{Let-Poly}\quad \frac{\Gamma \vdash v_1 : \tau_1 \qquad \bar{\alpha} \# \mathrm{ftv}(\tau_1) \qquad \Gamma; x : \forall \bar{\alpha}.\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = v_1\ \mathsf{in}\ e_2 : \tau_2} \qquad\qquad \textsc{Let-Mono}\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma; x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

Question 2 *On souhaite inférer le type du terme que nous avons évalué à la question 1. Étant données deux variables de types α et β, développer la définition de la contrainte*

$$[\![\lambda k.(\lambda x.x, \lambda f.k\ (f, \lambda d.5)) : (\alpha \to \beta) \to \alpha]\!]$$

Simplifier cette contrainte autant que possible. En déduire que le terme

$$\mathsf{callcc}\ (\lambda k.(\lambda x.x, \lambda f.k\ (f, \lambda d.5)))$$

admet tous les types (et uniquement les types) de la forme $(\tau \to \tau) \times ((\tau \to \tau) \to \mathsf{int})$. ◇

Solution. En appliquant les règles de génération de contraintes exposées en cours et en effectuant au vol quelques simplifications mineures, on vérifie que la contrainte

$$[\![\lambda k.(\lambda x.x, \lambda f.k\ (f, \lambda d.5)) : (\alpha \to \beta) \to \alpha]\!]$$

est équivalente à :

$$\mathsf{let}\ k : \alpha \to \beta\ \mathsf{in}\ \exists \alpha_1\alpha_2. \left(\begin{array}{l} \alpha = \alpha_1 \times \alpha_2 \wedge \\ \exists \alpha_3.\alpha_1 = \alpha_3 \to \alpha_3 \wedge \\ \exists \alpha_4\alpha_5. \left(\begin{array}{l} \alpha_2 = \alpha_4 \to \alpha_5 \wedge \\ \mathsf{let}\ f : \alpha_4\ \mathsf{in}\ \exists \alpha_6\alpha_7. \left(\begin{array}{l} k \preccurlyeq (\alpha_6 \times \alpha_7) \to \alpha_5 \wedge \\ f \preccurlyeq \alpha_6 \wedge \\ \exists \alpha_8.\alpha_8 \to \mathsf{int} = \alpha_7 \end{array} \right) \end{array} \right) \end{array} \right)$$

En extrudant tous les quantificateurs existentiels et en expansant les contraintes let, on obtient :

$$\exists \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6\alpha_7\alpha_8. \left(\begin{array}{l} \alpha = \alpha_1 \times \alpha_2 \wedge \\ \alpha_1 = \alpha_3 \rightarrow \alpha_3 \wedge \\ \alpha_2 = \alpha_4 \rightarrow \alpha_5 \wedge \\ \alpha \rightarrow \beta = (\alpha_6 \times \alpha_7) \rightarrow \alpha_5 \wedge \\ \alpha_4 = \alpha_6 \wedge \\ \alpha_8 \rightarrow \mathsf{int} = \alpha_7 \end{array}\right)$$

En décomposant la quatrième équation et en rapprochant les équations portant sur α, on obtient :

$$\exists \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6\alpha_7\alpha_8. \left(\begin{array}{l} \alpha = \alpha_1 \times \alpha_2 = \alpha_6 \times \alpha_7 \wedge \\ \alpha_1 = \alpha_3 \rightarrow \alpha_3 \wedge \\ \alpha_2 = \alpha_4 \rightarrow \alpha_5 \wedge \\ \beta = \alpha_5 \wedge \\ \alpha_4 = \alpha_6 \wedge \\ \alpha_8 \rightarrow \mathsf{int} = \alpha_7 \end{array}\right)$$

En décomposant la première équation et en rapprochant les deux équations portant sur α_1 d'une part, sur α_2 d'autre part, on obtient :

$$\exists \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6\alpha_7\alpha_8. \left(\begin{array}{l} \alpha = \alpha_1 \times \alpha_2 \wedge \\ \alpha_1 = \alpha_4 = \alpha_6 = \alpha_3 \rightarrow \alpha_3 \wedge \\ \alpha_2 = \alpha_7 = \alpha_8 \rightarrow \mathsf{int} = \alpha_4 \rightarrow \alpha_5 \wedge \\ \beta = \alpha_5 \end{array}\right)$$

En décomposant la troisième équation et en rapprochant les deux équations portant sur α_4 d'une part, sur α_5 d'autre part, on obtient :

$$\exists \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6\alpha_7\alpha_8. \left(\begin{array}{l} \alpha = \alpha_1 \times \alpha_2 \wedge \\ \alpha_1 = \alpha_4 = \alpha_6 = \alpha_8 = \alpha_3 \rightarrow \alpha_3 \wedge \\ \alpha_2 = \alpha_7 = \alpha_4 \rightarrow \alpha_5 \wedge \\ \beta = \alpha_5 = \mathsf{int} \end{array}\right)$$

En éliminant, par substitution, les variables existentiellement quantifiées dont la valeur est connue, il reste :

$$\exists \alpha_3. \left(\begin{array}{l} \alpha = (\alpha_3 \rightarrow \alpha_3) \times ((\alpha_3 \rightarrow \alpha_3) \rightarrow \mathsf{int}) \wedge \\ \beta = \mathsf{int} \end{array}\right)$$

Que dire à présent du terme callcc $(\lambda k.(\lambda x.x, \lambda f.k\ (f, \lambda d.5)))$? D'après la règle de typage de callcc, la contrainte

$$[\![\mathsf{callcc}\ (\lambda k.(\lambda x.x, \lambda f.k\ (f, \lambda d.5))) : \alpha]\!]$$

est équivalente à

$$\exists \beta.[\![\lambda k.(\lambda x.x, \lambda f.k\ (f, \lambda d.5)) : (\alpha \rightarrow \beta) \rightarrow \alpha]\!]$$

qui elle-même, d'après nos calculs précédents, est équivalente à

$$\exists \beta.\exists \alpha_3. \left(\begin{array}{l} \alpha = (\alpha_3 \rightarrow \alpha_3) \times ((\alpha_3 \rightarrow \alpha_3) \rightarrow \mathsf{int}) \wedge \\ \beta = \mathsf{int} \end{array}\right)$$

soit, en éliminant β,

$$\exists \alpha_3.\alpha = (\alpha_3 \rightarrow \alpha_3) \times ((\alpha_3 \rightarrow \alpha_3) \rightarrow \mathsf{int})$$

Ceci démontre que le terme callcc $(\lambda k. \ldots)$ admet tous les types, et uniquement les types, de la forme

$$(\tau \rightarrow \tau) \times ((\tau \rightarrow \tau) \rightarrow \mathsf{int})$$

□

Question 3 *Le terme de la question 1 :*

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{callcc}\ (\lambda k.(\lambda x.x, \lambda f.k\ (f, \lambda d.5)))\ \mathsf{in} \\ (\lambda z.\pi_2\ x\ (\lambda x.x+1))\ (\pi_1\ x\ \mathsf{true})\end{array}$$

est-il bien typé dans le système permissif ? Que dire alors de la sûreté du système permissif ? Ce terme est-il bien typé dans le système restreint ? (On demande une justification claire mais concise – ne construisez pas de dérivations de typage.) ◇

Solution. D'après la question précédente, le terme callcc $(\lambda k.\ldots)$ admet tous les types et uniquement les types de la forme $(\tau \to \tau) \times ((\tau \to \tau) \to \mathsf{int})$. Son schéma de types principal est donc

$$\forall \alpha.(\alpha \to \alpha) \times ((\alpha \to \alpha) \to \mathsf{int})$$

Dans le système de types permissif, on peut donc attribuer ce schéma à la variable (liée par let) x. On vérifie alors aisément que $\pi_2\ x$ admet (en particulier) le type $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ et que $\pi_1\ x$ admet (en particulier) le type $\mathsf{bool} \to \mathsf{bool}$, ce qui permet de conclure que le terme complet admet le type int. Ce terme est donc bien typé dans le système permissif. Or, d'après la question 1, sa réduction conduit à un terme bloqué. Le système permissif n'est donc pas sûr.

Dans le système de types restreint, le type attribué à callcc $(\lambda k.\ldots)$ ne peut pas être généralisé, car cette expression n'est pas une valeur. Le type attribué à la variable (liée par let) est donc de la forme $(\tau \to \tau) \times ((\tau \to \tau) \to \mathsf{int})$, pour un certain type τ. Mais alors l'expression $\pi_2\ x\ (\lambda x.x+1)$ exige $\tau = \mathsf{int}$, tandis que l'expression $\pi_1\ x\ \mathsf{true}$ exige $\tau = \mathsf{bool}$. Par conséquent, le terme complet est mal typé dans le système restreint. □

Question 4 *Démontrer, dans le système restreint, la propriété suivante, et indiquer pourquoi elle est fausse dans le système permissif. Supposons que le jugement $\vdash E[e] : \tau_0$ soit dérivable à partir du jugement $\vdash e : \tau_1$. Supposons de plus que e n'est pas une valeur. Alors le jugement $x : \tau_1 \vdash E[x] : \tau_0$ est dérivable.* ◇

Solution. La preuve se fait par induction sur la structure du contexte d'évaluation E.

∘ *Cas* E *est un trou* $[\,]$. L'hypothèse devient : le jugement $\vdash e : \tau_0$ est dérivable à partir du jugement $\vdash e : \tau_1$. L'expression e qui apparaît dans ces deux jugements étant la même, ils sont alors nécessairement identiques. Il en résulte que τ_1 est égal à τ_0. Le jugement $x : \tau_1 \vdash x : \tau_0$ est donc dérivable grâce à la règle de typage des variables.

∘ *Cas* E est de la forme let $x_0 = E_0$ in e_0. L'hypothèse devient : le jugement $\vdash \mathsf{let}\ x_0 = E_0[e]\ \mathsf{in}\ e_0 : \tau_0$ est dérivable à partir du jugement $\vdash e : \tau_1$. La dernière règle appliquée dans cette dérivation est nécessairement l'une de celles qui concernent la construction let. De plus, e n'étant pas une valeur, $E_0[e]$ n'est pas non plus une valeur. Dans le système de types restreint, c'est donc nécessairement la règle Let-Mono qui a été appliquée. La dérivation se termine ainsi par une inférence de la forme :

$$\frac{\vdash E_0[e] : \tau_0' \qquad x_0 : \tau_0' \vdash e_0 : \tau_0}{\vdash \mathsf{let}\ x_0 = E_0[e]\ \mathsf{in}\ e_0 : \tau_0}$$

où la prémisse $\vdash E_0[e] : \tau_0'$ est dérivable à partir du jugement $\vdash e : \tau_1$. D'après l'hypothèse d'induction, le jugement $x : \tau_1 \vdash E_0[x] : \tau_0'$ est donc dérivable. De plus, la prémisse $x_0 : \tau_0' \vdash e_0 : \tau_0$ implique, par renforcement de l'environnement, $x : \tau_1; x_0 : \tau_0' \vdash e_0 : \tau_0$. On conclut alors par application de la règle Let-Mono :

$$\frac{x : \tau_1 \vdash E_0[x] : \tau_0' \qquad x : \tau_1; x_0 : \tau_0' \vdash e_0 : \tau_0}{x : \tau_1 \vdash \mathsf{let}\ x_0 = E_0[x]\ \mathsf{in}\ e_0 : \tau_0}$$

Cette propriété est fausse dans le système permissif car la règle de typage de la construction let permet alors la généralisation. La dérivation considérée se termine alors par une inférence de forme plus générale :

$$\frac{\vdash E_0[e] : \tau_0' \qquad x_0 : \forall \bar{\alpha}.\tau_0' \vdash e_0 : \tau_0}{\vdash \mathsf{let}\ x_0 = E_0[e]\ \mathsf{in}\ e_0 : \tau_0}$$

où aucune condition ne porte sur $\bar{\alpha}$, puisque l'environnement de typage est vide. Pour pouvoir, comme ci-dessus, introduire l'hypothèse $x : \tau_1$ et conclure, il faudrait pouvoir affirmer $\bar{\alpha} \# \mathrm{ftv}(\tau_1)$, ce qui est faux en général. □

Question 5 *À l'aide de la propriété précédente, démontrer que, dans le système restreint, la réduction préserve le type* unit*, c'est-à-dire que* $\vdash e : \mathsf{unit}$ *et* $e \to e'$ *impliquent* $\vdash e' : \mathsf{unit}$*. On s'intéressera uniquement au cas des règles de réduction (3) et (4), le cas des règles (1) et (2) étant sans intérêt particulier.* ◇

Solution. ∘ *Cas* de la règle (3). Nous avons, par hypothèse, $\vdash E[\mathsf{callcc}\ v] : \mathsf{unit}$. La dérivation de ce jugement s'appuie (indirectement) sur une prémisse de la forme $\vdash \mathsf{callcc}\ v : \tau_1$ (**1**), pour un certain τ_1. Le terme $\mathsf{callcc}\ v$ n'étant pas une valeur, nous avons, d'après la propriété précédente :

$$x : \tau_1 \vdash E[x] : \mathsf{unit}$$

En appliquant la règle de typage associée à abort, nous obtenons alors

$$x : \tau_1 \vdash \mathsf{abort}\ E[x] : \tau_2$$

puis, en appliquant la règle de typage associée aux abstractions,

(**2**) $$\vdash \lambda x.\mathsf{abort}\ E[x] : \tau_1 \to \tau_2$$

Considérons à nouveau le jugement (1). D'après la règle de typage associée à callcc, il est nécessairement dérivé d'un jugement de la forme

(**3**) $$\vdash v : (\tau_1 \to \tau_2) \to \tau_1$$

pour un certain τ_2. En combinant (3) et (2), nous obtenons ainsi

(**4**) $$\vdash v\ (\lambda x.\mathsf{abort}\ E[x]) : \tau_1$$

Il ne reste plus qu'à remarquer que, si de (1) on a pu déduire $\vdash E[\mathsf{callcc}\ v] : \mathsf{unit}$, alors de (4) on déduit, de façon identique,

$$\vdash E[v\ (\lambda x.\mathsf{abort}\ E[x])] : \mathsf{unit}$$

∘ *Cas* de la règle (4). Nous avons, par hypothèse, $\vdash E[\mathsf{abort}\ e] : \mathsf{unit}$. La dérivation de ce jugement s'appuie (indirectement) sur une prémisse de la forme $\vdash \mathsf{abort}\ e : \tau$, qui elle-même, d'après la règle de typage associée à abort, dérive nécessairement du jugement $\vdash e : \mathsf{unit}$. Or c'est précisément notre but. □

Il resterait, avant de pouvoir conclure à la sûreté du système de types restreint, à vérifier le lemme de progrès. Cela se fait sans difficulté particulière.