COMPOSITION D'INFORMATIQUE n°6

Corrigé

⋆ ⋆ ⋆

1 Inférence d'automates

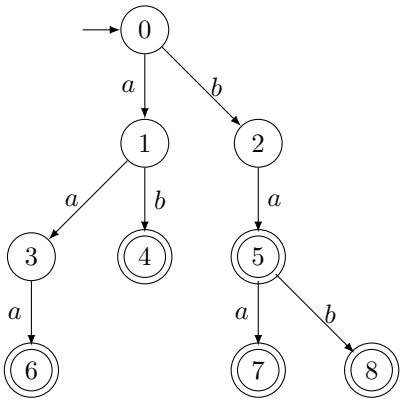
Question 1 Il s'agit d'un algorithme de classification en apprentissage supervisé : on dispose d'un jeu de données étiquetées par deux classes (positif et négatif), et on souhaite pouvoir classifier n'importe quelle nouvelle donnée.

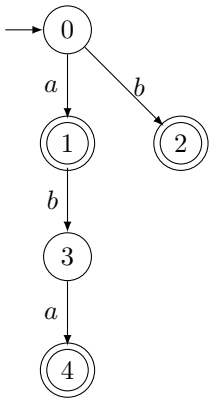
Question 2 Soit A_+ un automate fini tel que $L_+ = L(A_+)$. Alors $L_+ \subseteq L(A_+)$ (car il y a égalité) et $L_- \cap L(A_+) = \emptyset$ (car L_+ et L_- sont disjoints). De même, soit $\overline{A_-}$ un automate fini reconnaissant $\overline{L_-}$. Comme $L_+ \cap L_- = \emptyset$, alors $L_+ = L_+ \cap \overline{L_-} \subseteq L(\overline{A_-})$. De plus, $L_- \cap L(\overline{A_-}) = \emptyset$ par définition du complémentaire.

Question 3 Il peut y avoir un problème de surapprentissage si on souhaite trop coller aux données de l'échantillon : un automate reconnaissant L_+ par exemple ne reconnaîtrait aucun autre mot.

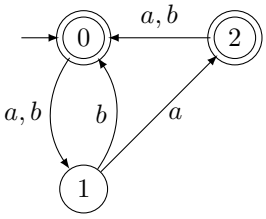
Question 4 On remarque que si $L = \emptyset$ ou $L = \{\varepsilon\}$, alors $k = 0$, et on aurait $A = (\{q_0\}, \Delta, q_0, \emptyset)$ d'une part et $A = (\{q_0\}, \Delta, q_0, \{q_0\})$ d'autre part, où $\Delta(q_0, a) = \emptyset$. Ces cas servent d'initialisation.

Question 5 On obtient les automates arborescents suivants :





Question 6 L'automate suivant convient :



Question 7 Par l'absurde, supposons qu'il existe un automate déterministe complet $A = (\{q_0, q_1\}, \Delta, q_0, F)$ inféré par L_+ et L_-.

- Comme l'automate est complet et que $L(A) \neq \Sigma^*$ et $\neq \emptyset$, l'un des état est final et l'autre non ;
- comme $a, b \in L_-$, on en déduit que $\Delta(q_0, a) = \Delta(q_0, b)$ est l'état non final ;
- comme $aaa \in L_+$, on en déduit que $\Delta(q_0, a) = \Delta(q_0, b) = \{q_1\}$ (sinon a et aaa permettrait d'atteindre le même état q_0) ;

– comme $ab \in L_+$ et $ba \in L_+$, on en déduit que $\Delta(q_1, a) = \Delta(q_1, b) = \{q_0\}$.

Mais alors $\Delta^*(q_0, aaa) = \{q_1\}$ n'est pas un état final, alors que $aaa \in L_+$. On conclut par l'absurde.

Question 8 On montre ce résultat par récurrence forte sur la taille maximale n d'un mot de L. Le cas $L = \emptyset$ est traité à part.

– si $n = 0$, alors $L = \{\varepsilon\}$ et on a vu en question 4 que $L = L(T(L))$;

– supposons le résultat établi jusqu'à $n \in \mathbb{N}$ fixé et soit L un langage fini dont le mot le plus long est de taille $n+1$. En utilisant les notations de l'énoncé, par hypothèse de récurrence, $L_i = L(A_i)$, car le mot le plus long d'un des L_i est de taille n. De plus, par définition, $L = (L \cap \varepsilon) \cup \bigcup_{i=1}^{k} L \cap a_i L_i$. Montrons que $L = L(A)$ où $A = (Q, \Delta, q_0, F)$ est l'automate $T(L)$:

  * si $u = \varepsilon$, alors $u \in L \Leftrightarrow q_0 \in F \Leftrightarrow u \in L(A)$;
  * sinon, si $u = av$, alors :
    — si $u \in L$, alors $a = a_i$ pour un certain $i \in [\![1, k]\!]$ et $v \in L_i$. Par hypothèse, $\Delta_i^*(q_i, v) \cap F_i \neq \emptyset$. Comme $\Delta(q_0, a) = \{q_i\}$, on en déduit que $\Delta^*(q_0, u) \cap F \neq \emptyset$, donc $u \in L(A)$;
    — si $u \in L(A)$, alors il existe un calcul acceptant de u dans A ; comme $\Delta(q_0, \cdot)$ n'est défini que sur $\{a_1, \ldots, a_k\}$, on en déduit que $a = a_i$ pour un certain $i \in [\![1, k]\!]$ et que $\Delta_i^*(q_i, v) \cap F_i \neq \emptyset$. Par hypothèse de récurrence, cela montre que $v \in L_i$ et donc que $u = a_i v \in L$.

On conclut par récurrence.

Question 9 Il s'agit ici de faire les bonnes allocations mémoire et d'initialiser le tableau de transitions avec des pointeurs `NULL`.

```
etat* creer_etat(void){
    etat* q = malloc(sizeof(*q));
    q->final = false;
    q->Delta = malloc(N * sizeof(etat*));
    for (int i=0; i<N; i++){
        q->Delta[i] = NULL;
    }
    return q;
}
```

Question 10 On ne fait une libération que si l'état est non `NULL`. Lorsque c'est le cas, on libère récursivement tous les états qui sont accessibles, avant de libérer le tableau et l'état.

```
void liberer_etat(etat* q){
    if (q != NULL){
        for (int i=0; i<N; i++){
            liberer_etat(q->Delta[i]);
        }
        free(q->Delta);
        free(q);
    }
}
```

Question 11 Telle quelle, la fonction ne terminerait pas si on tombe sur un cycle, car on fait l'appel récursif avant de libérer le pointeur. Selon la manière dont est écrite la fonction, il est également possible d'avoir les problèmes suivants :

– appel à `free` sur un pointeur déjà libéré ;

– calcul de `q->Delta` pour `q` un pointeur déjà libéré.

Question 12 On garde en mémoire l'état courant, initialisé avec `q0`. À chaque itération, on lit une transition d'une nouvelle lettre de u, et si cette transition n'est pas définie, c'est que le mot n'est pas reconnu. Une fois le mot lu, on vérifie si l'état qu'on a atteint est final ou non.

```
bool reconnu(etat* q0, int* u, int n){
    etat* q = q0;
    for (int i=0; i<n; i++){
        q = q->Delta[u[i]];
        if (q == NULL){
            return false;
        }
    }
    return q->final;
}
```

Cette fonction aurait pu être codée récursivement :

```
bool reconnu(etat* q, int* u, int n){
    if (q == NULL){
        return false;
    }else if (n == 0){
        return q->final;
    }else{
        return reconnu(q->Delta[u[0]], &u[1], n - 1);
    }
}
```

Question 13 On commence par écrire une fonction qui ajoute la reconnaissance d'un mot dans un automate fini.

```
void ajouter_mot(etat* q0, int* u, int n){
    if (n == 0){
        q0->final = true;
        return;
    }
    if (q0->Delta[u[0]] == NULL){
        etat* q = creer_etat();
        q0->Delta[u[0]] = q;
    }
    ajouter_mot(q0->Delta[u[0]], &u[1], n - 1);
}
```

L'idée est que si le mot est vide, il suffit de rendre final l'état où on est arrivé, sinon on supprime la première lettre et on ajoute le mot depuis l'état atteint en lisant cette première lettre. On note qu'on crée l'état s'il n'existe pas déjà.

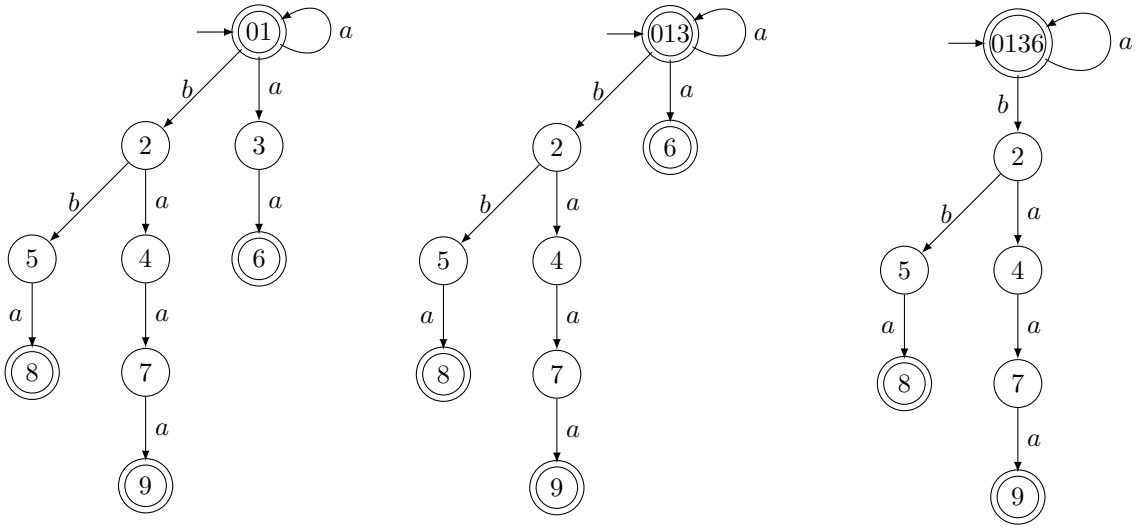
Dès lors, il suffit d'ajouter tous les mots du langage un par un depuis un état unique :

```
etat* arborescent(int** L, int* tailles, int k){
    etat* q0 = creer_etat();
    for (int i=0; i<k; i++){
        ajouter_mot(q0, L[i], tailles[i]);
    }
    return q0;
}
```

Question 14 On détaille :

- la fonction `creer_etat` a une complexité linéaire en N car il faut créer le tableau de transitions ;
- la fonction `ajouter_mot` a une complexité en $\mathcal{O}(N \times |u|)$ pour ajouter un mot u. Cette complexité n'est pas toujours atteinte, notamment si les états existent déjà ;
- la fonction `arborescent` est donc en $\mathcal{O}(|\Sigma| \times n_{\max} \times |L|)$ où $n_{\max}$ est la longueur maximale d'un mot de L.

Question 15 On représente les contractions successives :



Le dernier automate est alors déterministe, donc l'algorithme termine.

Question 16 L'opération de contraction termine bien en temps fini. Le corps de la boucle **Tant que** est fini (parcourir l'automate, choisir des éléments et faire une contraction). Comme le nombre d'états de l'automate A' diminue strictement à chaque itération, c'est un variant de boucle. De plus, quand $|Q'| = 1$, l'algorithme termine, car un automate avec un seul état est toujours déterministe. On conclut que `DetMerge` termine.

Question 17 Soit $u = a_1 \dots a_n \in L(A)$ et $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \dots \xrightarrow{a_n} q_n$ un calcul réussi de u dans A (c'est-à-dire tel que $q_i \in \Delta(q_{i-1}, a_i)$ pour $i \in [\![1, n]\!]$ et $q_n \in F$). On note r l'état qui résulte de la fusion de p et q dans $A/\{p, q\}$. On pose, pour $i \in [\![0, n]\!]$:

$$q'_i = \begin{cases} q_i & \text{si } q_i \notin \{p, q\} \\ r & \text{sinon} \end{cases}$$

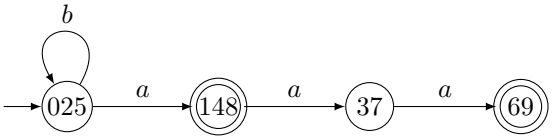
Alors $q'_0 \xrightarrow{a_1} q'_1 \xrightarrow{a_2} q'_2 \dots \xrightarrow{a_n} q'_n$ est un calcul réussi de u dans $A/\{p, q\} = (Q', \Delta', q'_0, F')$. En effet, si $q'_n = q_n$, alors $q'_n \in F'$, et sinon $q_n \in \{p, q\}$, donc $q'_n = r \in F'$. De plus, par définition de la contraction, si $q_i \in \Delta(q_{i-1}, a_i)$, alors $q'_i \in \Delta'(q'_{i-1}, a_i)$.

On en conclut que $u \in L(A/\{p, q\})$.

Finalement, comme $\texttt{DetMerge}(A, p, q)$ est obtenu par contraction successives dans le même automate, par transitivité de l'inclusion, on en déduit bien $L(A) \subseteq L(\texttt{DetMerge}(A, p, q))$.

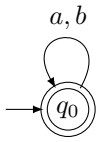
Question 18 Dans l'automate $\texttt{DetMerge}(A, 0, 1)$ calculé à une question précédente, on constate que le mot *aaaa* est reconnu, alors que c'est un mot de L_-. On en déduit que la procédure ne peut pas continuer.

Question 19 On repart de l'automate :

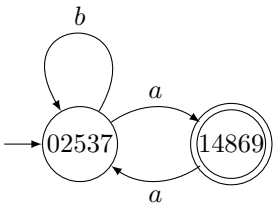


On a :

– en faisant la fusion déterministe de 025 et 148, on obtient un automate avec un unique état qui reconnait Σ^*. Cet automate ne permet pas de continuer RPNI, donc on annule la fusion.



– en faisant la fusion déterministe de 025 et 37, on obtient l'automate :



Cet automate ne reconnait aucun mot de L_-, donc on peut le conserver.

– si on lance la procédure `DetMerge` sur les deux états restants, on obtient à nouveau un automate qui reconnait tous les mots, donc ne convient pas. L'algorithme RPNI termine donc.

Question 20 Montrons que « A est un automate inféré par L_+ et L_- » est un invariant de la boucle **Tant que** dans l'algorithme RPNI :

– c'est vrai avant la boucle par les questions 2 et 8 ;

– supposons que c'est vrai au début d'un passage dans la boucle et distinguons :

* si A n'a pas été modifié, alors le résultat reste vrai à la fin du passage ;
* sinon, A a été modifié en A'. Or, par la question 17, $L_+ \subseteq L(A) \subseteq L(A')$. De plus, par le test, $L(A') \cap L_- = \emptyset$. On en déduit que A' est un automate inféré par L_+ et L_-.

Le résultat reste donc vrai à la sortie de la boucle, ce qui conclut.

Question 21 On détaille chaque opération :

– pour un automate $A = (Q, \Delta, q_0, F)$, l'opération de contraction peut se faire en temps $\mathcal{O}(|\Delta|+|Q|)$ où $|\Delta|$ est un abus de notation pour indiquer le nombre total de transitions : on peut utiliser une représentation d'un automate sous forme de graphe. La contraction consiste alors à créer un nouveau graphe, de taille similaire, en distinguant selon qu'une transition concerne un état contracté ou non. ;

– à ce titre, l'opération `DetMerge` nécessite de trouver deux états dans un même $\Delta(r, a)$. Cela peut se faire en temps linéaire en la taille de l'automate (états et transitions compris), donc ce n'est pas dominant par rapport à la contraction. Dans le pire cas, on fait de l'ordre de $\mathcal{O}(|Q|)$ contraction, soit une complexité de `DetMerge` en $\mathcal{O}(|Q|(|Q| + |\Delta|))$, car le nombre total de transitions ne peut que diminuer ;

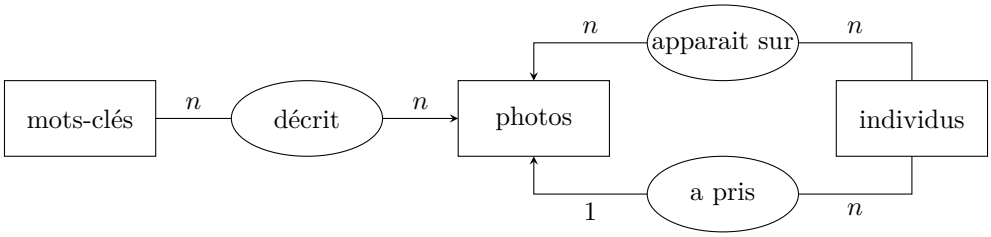
– dans l'algorithme RPNI, on doit :

* faire le calcul de $T(L_+)$, qui peut se faire en $\mathcal{O}(n_{\max} \times |L_+|) = \mathcal{O}(|Q|)$ où $n_{\max}$ est la longueur maximale d'un mot de L_+, avec la représentation sous forme de graphe (il n'est pas nécessaire de créer un tableau de taille $|\Sigma|$) ;
* pour chacun des $\mathcal{O}(|Q|^2)$ passages dans la boucle **Tant que**, faire un appel à `DetMerge` et tester s'il existe un mot de L_- qui est reconnu (ce qui se fait en $\mathcal{O}(n_{\max} \times |L_-|)$).

La complexité totale est donc en $\mathcal{O}(|Q|^2(|Q|(|Q| + |\Delta|) + n_{\max} \times |L_-|))$. Or le nombre de transitions dans $T(L_+)$ est de l'ordre de $\mathcal{O}(|Q|) = \mathcal{O}(n_{\max} \times |L_+|)$. En supposant que $|L_+|$ est du même ordre de grandeur que $|L_-|$, on a donc $\mathcal{O}((n_{\max}|L_+|)^4)$.

2 Base de données photographique

Question 22 On considère trois entités : les individus, les photos et les mots-clés. On représente le diagramme de la forme :



Question 23 La paire `(inId, phId)` constitue la clé primaire de la table `Présent` : chacun de ces deux attributs pris isolément n'est pas une clé primaire. Il est donc possible d'avoir deux enregistrements avec la même valeur pour l'attribut `inId`, donc la même personne présente sur deux photos. De même plusieurs personnes différentes peuvent apparaitre sur une même photo.

Question 24 On a :

```
SELECT phId FROM Photos
WHERE phLarg / phHaut = 16 / 9
```

Question 25 On propose :

```
SELECT inNom FROM Individus AS in
JOIN Présent AS pr ON pr.inId = in.inId
JOIN Photos AS ph ON ph.phId = pr.phId
WHERE phAuteur = in.inId
```

Question 26 On propose :

```
SELECT phId FROM Photos
WHERE phId NOT IN
   (SELECT phId FROM Présent)
```

Question 27 On obtient :

```
SELECT phFichier FROM Photos AS ph
JOIN Description AS de ON ph.phId = de.phId
GROUP BY ph.phId
HAVING COUNT(motCle) >= 5
```

3 Logique

Question 28 On pose $\Gamma = \{\varphi, \neg\varphi\}$:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \varphi}\,\text{ax} \qquad \dfrac{}{\Gamma \vdash \neg\varphi}\,\text{ax}}{\Gamma \vdash \bot}\,\neg_e}{\varphi \vdash \neg\neg\varphi}\,\neg_i$$

Question 29 On a la preuve :

$$\dfrac{\dfrac{}{\neg(\varphi\vee\psi),\varphi\vdash\neg(\varphi\vee\psi)}\,\text{ax}\qquad\dfrac{\dfrac{}{\neg(\varphi\vee\psi),\varphi\vdash\varphi}\,\text{ax}}{\neg(\varphi\vee\psi),\varphi\vdash\varphi\vee\psi}\,\vee_i}{\dfrac{\dfrac{\neg(\varphi\vee\psi),\varphi\vdash\bot}{\neg(\varphi\vee\psi)\vdash\neg\varphi}\,\neg_i\qquad\qquad\dfrac{\text{idem}}{\neg(\varphi\vee\psi)\vdash\neg\psi}}{\neg(\varphi\vee\psi)\vdash\neg\varphi\wedge\neg\psi}\,\vee_i}\,\neg_e$$

Question 30 On pose $\Gamma=\{\forall x\,\varphi,\exists x\,\neg\varphi\}$. On a la preuve :

$$\dfrac{\dfrac{}{\Gamma\vdash\exists x\,\neg\varphi}\,\text{ax}\qquad\dfrac{\dfrac{}{\Gamma,\neg\varphi\vdash\neg\varphi}\,\text{ax}\qquad\dfrac{\dfrac{}{\Gamma,\neg\varphi\vdash\forall x\,\varphi}\,\text{ax}}{\Gamma,\neg\varphi\vdash\varphi}\,\forall_e}{\Gamma,\neg\varphi\vdash\bot}\,\neg_e}{\dfrac{\Gamma\vdash\bot}{\forall x\,\varphi\vdash\neg(\exists x\,\neg\varphi)}\,\neg_i}\,\exists_e$$

L'élimination du quantificateur existentiel est correcte car x est libre dans Γ et dans $\bot$.

Question 31 On a le jugement de typage $\vdash$ `List.map (fun x -> x + 1) : int list -> int list`

Question 32 On a la dérivation suivante :

$$\dfrac{\dfrac{}{\Gamma,x:\beta\vdash g:\beta\to\alpha}\,\text{C}\qquad\dfrac{}{\Gamma,x:\beta\vdash x:\beta}\,\text{C}}{\Gamma,x:\beta\vdash g\ x:\alpha}\,\to_e$$

Question 33 On obtient alors :

$$\dfrac{\dfrac{\dfrac{\text{Question précédente}}{\Gamma,x:\beta\vdash g\ x:\alpha}\qquad\dfrac{}{\Gamma,x:\beta\vdash f:\alpha\to(\beta\to\gamma)}\,\text{C}}{\Gamma,x:\beta\vdash f\ (g\ x):\beta\to\gamma}\,\to_e\qquad\dfrac{}{\Gamma,x:\beta\vdash x:\beta}\,\text{C}}{\Gamma,x:\beta\vdash f\ (g\ x)\ x:\gamma}\,\to_e}{\Gamma\vdash(\texttt{fun}\ x\to f\ (g\ x)\ x):\beta\to\gamma}\,\to_i$$

Question 34 L'expression e n'est ni une constante du langage, ni une variable, ni de la forme $\texttt{fun}\ x\to e'$. On en déduit que dans une dérivation de jugement de e, la dernière règle à utiliser serait nécessairement $(\to_e)$. On aurait donc un arbre de dérivation dont la racine et ses deux enfants sont de la forme :

$$\dfrac{\Gamma\vdash(\texttt{fun}\ h\to h\ 1\ 2):\alpha\to\tau\qquad\Gamma\vdash(\texttt{fun}\ x\to 3):\alpha}{\Gamma\vdash(\texttt{fun}\ h\to h\ 1\ 2)\ (\texttt{fun}\ x\to 3):\tau}\,\to_e$$

où les prémisses peuvent être dérivées.

Question 35 Pour des raisons similaires à la question précédente, la dernière règle à utiliser serait nécessairement $(\to_i)$. On aurait alors :

$$\dfrac{\Gamma,x:\sigma\vdash 3:\beta}{\Gamma\vdash(\texttt{fun}\ x\to 3):\sigma\to\beta}\,\to_i$$

On en déduit que $\alpha=\sigma\to\beta$. De plus, sachant que la seule règle permettant de typer 3 est $\dfrac{}{\Gamma,x:\sigma\vdash 3:\texttt{int}}\,\text{ax}$, on en déduit que $\alpha=\sigma\to\texttt{int}$.

Question 36 À nouveau, seule la règle $(\to_i)$ a pu s'appliquer. On aurait alors :

$$\dfrac{\Gamma,h:\sigma\to\texttt{int}\vdash h\ 1\ 2:\tau}{\Gamma\vdash(\texttt{fun}\ h\to h\ 1\ 2):(\sigma\to\texttt{int})\to\tau}\,\to_i$$

En continuant, on ne peut appliquer que la règle $(\rightarrow_e)$, et on obtient :

$$\frac{\Gamma, h : \sigma \rightarrow \texttt{int} \vdash h\ 1 : \gamma \rightarrow \tau \qquad \Gamma, h : \sigma \rightarrow \texttt{int} \vdash 2 : \gamma}{\Gamma, h : \sigma \rightarrow \texttt{int} \vdash h\ 1\ 2 : \tau} \rightarrow_e$$

On en déduit que $\gamma = \texttt{int}$, ce qui permet de remonter sur la prémisse de gauche, à nouveau uniquement avec $(\rightarrow_e)$:

$$\frac{\Gamma, h : \sigma \rightarrow \texttt{int} \vdash h : \delta \rightarrow (\texttt{int} \rightarrow \tau) \qquad \Gamma, h : \sigma \rightarrow \texttt{int} \vdash 1 : \delta}{\Gamma, h : \sigma \rightarrow \texttt{int} \vdash h\ 1 : \texttt{int} \rightarrow \tau} \rightarrow_e$$

À nouveau, $\delta = \texttt{int}$. Finalement, comme h est une variable, son seul type possible est $\sigma \rightarrow \texttt{int}$. Cela implique que $\sigma = \texttt{int}$ et que $\texttt{int} = \texttt{int} \rightarrow \tau$, ce qui est absurde d'après l'hypothèse donnée dans l'énoncé.

$\star\star\star$