Terminaison des systèmes concurrents d'ordre supérieur

Stage de L3 effectué sous la direction de Daniel Hirschkoff

Simon Castellan

22 janvier 2014

Introduction

- étude des langages concurrents du point de vue de la terminaison ;
- principale contribution du stage : $\lambda\pi$, un mélange de λ-calcul et π-calcul : description simultanée des aspects séquentiels et concurrents.
- ...tout en garantissant la terminaison via un système de types.

Plan de l'exposé

Plan de l'exposé

$\lambda\pi$ comme un langage de programmation

- cadre séquentiel (*fonctions*) :
  - définition de fonctions, passage en paramètre de fonctions, contient en fait tout le λ-calcul
- cadre concurrent (*processus*)
  - canaux sur lesquels on peut envoyer des messages
  - envoi d'un message t sur un canal a, $\overline{a}\langle t\rangle$
  - réception d'une valeur k sur a puis exécution de P : $a(k).P$
  - exécution parallèle de deux processus (ou *threads*) $P \parallel Q$

$$\texttt{let } S_0 = a(k).\ (k\ ()\ \parallel\ \overline{a}\langle k\rangle) \texttt{ in } S_0 \parallel \overline{a}\langle \texttt{fun } x \to S_0\rangle$$

$\lambda\pi$ comme un langage de programmation

- cadre séquentiel (*fonctions*) :
  - définition de fonctions, passage en paramètre de fonctions, contient en fait tout le λ-calcul
- cadre concurrent (*processus*)
  - canaux sur lesquels on peut envoyer des messages
  - envoi d'un message t sur un canal a, $\overline{a}\langle t\rangle$
  - réception d'une valeur k sur a puis exécution de P : $a(k).P$
  - exécution parallèle de deux processus (ou *threads*) $P \parallel Q$

$$\texttt{let } S_0 = \\ \quad a(k).\ (k\ ()\ \texttt{||}\ \overline{a}\langle k\rangle) \\ \texttt{in } S_0\ \texttt{||}\ \overline{a}\langle \texttt{fun } x \to S_0\rangle$$

$\lambda\pi$ comme un langage de programmation

- cadre séquentiel (*fonctions*) :
  - définition de fonctions, passage en paramètre de fonctions, contient en fait tout le λ-calcul
- cadre concurrent (*processus*)
  - canaux sur lesquels on peut envoyer des messages
  - envoi d'un message t sur un canal a, $\overline{a}\langle t\rangle$
  - réception d'une valeur k sur a puis exécution de P : $a(k).P$
  - exécution parallèle de deux processus (ou *threads*) $P \parallel Q$

$$\texttt{let } S_0 = a(k).\ (k\ ()\ \texttt{||}\ \overline{a}\langle k\rangle) \texttt{ in } S_0\ \texttt{||}\ \overline{a}\langle \texttt{fun } x \to S_0\rangle$$

$\lambda\pi$ comme un langage de programmation

- cadre séquentiel (*fonctions*) :
  - définition de fonctions, passage en paramètre de fonctions, contient en fait tout le λ-calcul
- cadre concurrent (*processus*)
  - canaux sur lesquels on peut envoyer des messages
  - envoi d'un message t sur un canal a, $\overline{a}\langle t\rangle$
  - réception d'une valeur k sur a puis exécution de P : $a(k).P$
  - exécution parallèle de deux processus (ou *threads*) $P \parallel Q$

$$\texttt{let } S_0 = a(k).\ (k\ ()\ \parallel\ \overline{a}\langle k\rangle) \texttt{ in } S_0\ \parallel\ \overline{a}\langle \texttt{fun } x \to S_0\rangle$$

Quelques exemples de programmes : ordre supérieur

- l'envoi de fonctions permet d'exécuter du code sur un nœud distant ;
- par exemple

```
wikipédia <fun () -> let résultat = ... in
                     local<résultat> >
|| local(x). print x
```

- calcul de `résultat` sur le nœud `wikipédia`
- renvoi du résultat à la source

Quelques exemples de programmes : ordre supérieur

- l'envoi de fonctions permet d'exécuter du code sur un nœud distant ;
- par exemple

```
wikipédia <fun () -> let résultat = ... in
                     local<résultat> >
|| local(x). print x
```

- calcul de `résultat` sur le nœud `wikipédia`
- renvoi du résultat à la source

Quelques exemples de programmes : ordre supérieur

- l'envoi de fonctions permet d'exécuter du code sur un nœud distant ;
- par exemple

```
wikipédia <fun () -> let résultat = ... in
                     local<résultat> >
|| local(x). print x
```

- calcul de `résultat` sur le nœud `wikipédia`
- renvoi du résultat à la source

Quelques exemples de programmes : ordre supérieur

- l'envoi de fonctions permet d'exécuter du code sur un nœud distant ;
- par exemple

```
wikipédia <fun () -> let résultat = ... in
                     local<résultat> >
|| local(x). print x
```

- calcul de `résultat` sur le nœud `wikipédia`
- renvoi du résultat à la source

Quelques exemples de programmes : divergences

- à chaque communication, le terme qui écoute est mangé
- difficile de faire des serveurs ou des termes divergents
- l'exemple précédent :

  $\texttt{let } S_0 = \quad a(k).\ (k\ ()\ \texttt{||}\ \overline{a}\langle k\rangle)$
  $\texttt{in } S_0 \quad \texttt{||}\ \overline{a}\langle \texttt{fun } x \to S_0\rangle$

  est divergent
- analogue concurrent de l'auto-application en λ-calcul.

Quelques exemples de programmes : divergences

- à chaque communication, le terme qui écoute est mangé
- difficile de faire des serveurs ou des termes divergents
- l'exemple précédent :

  $\texttt{let } S_0 = \quad a(k).\ (k\ ()\ \texttt{||}\ \overline{a}\langle k\rangle)$
  $\texttt{in } S_0 \quad \texttt{||}\ \overline{a}\langle \texttt{fun } x \to S_0\rangle$

  est divergent
- analogue concurrent de l'auto-application en λ-calcul.
- au contraire ce programme

  $\texttt{let } S\ ()\ = a(k).(k\ ()\ \texttt{||}$
  $b(x).(\overline{c}\langle x\rangle\ \texttt{||}\ \overline{a}\langle k\rangle))\ \texttt{in}$
  $S\ ()\ \texttt{||}\ \overline{a}\langle S\rangle$

  termine et représente un serveur qui redirige le traffic de b vers c.

Concurrence et distribution : passivation

- $\overline{a}\langle t \rangle$ peut se réduire si t le peut.
- $\overline{a}\langle P \rangle$ peut signifier "P s'exécute à la localité a"
- dans ce cas, $a(X).Q$ signifie d'ôter le processus s'exécutant à la localité a et d'exécuter Q : mise à jour : $a(X).\overline{a}\langle L \rangle$
- envoi d'un message à l'intérieur d'une localité : $a(X).\overline{a}\langle X \mid\mid \overline{b}\langle t \rangle \rangle$

Évaluation d'un programme

- règle primitive d'évaluation des programmes séquentiels :
  $(\texttt{fun}\ x \to t)\ u \rightsquigarrow t[u/x]$
- règle primitive d'évaluation des programmes concurrents : une réception en parallèle avec un envoi :
  $a(x).P \mid\mid \overline{a}\langle t \rangle \rightsquigarrow P[t/x]$
- formes normales, calculs aboutis ou en attente de l'environnement :
  - $\texttt{fun}\ x \to t$;
  - $a(x).P$;

Plan de l'exposé

Retour sur les divergences

- $a(k).(\overline{a}\langle k\rangle \parallel k\ ()) \parallel \overline{a}\langle\ldots\rangle$ diverge ;
- cycles : $a \to b \parallel b \to a$
- ordre bien fondé sur les canaux : chaque canal a un niveau ($\in \mathbb{N}$)
- $a(x).P$ *valide* ($=$ accepté par les règles de typage) seulement si pas d'envois supérieur à a dans P

Typage en $\lambda\pi$ (fonctions)

- terminaison : propriété indécidable $\Rightarrow$ approximation au travers d'un système de types
  - jugement de typage $\Gamma \vdash t : \tau$
  - deux composantes à typer : fonctions et canaux
- type des fonctions : type de l'argument $(\sigma) \rightarrow$ type de retour (τ)

  $(\texttt{let}\ f = (\texttt{fun}\ x \rightarrow x > 0))$: int$\rightarrow$bool.
- attribution du type à une fonction,

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash (\texttt{fun}\ x \rightarrow t) : \sigma \rightarrow \tau}$$

- application : le type de l'argument doit correspondre au type attendu par la fonction

$$\frac{\Gamma \vdash t : \sigma \rightarrow \tau \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash t\ u : \tau}$$

Typage en $\lambda\pi$ (canaux)

- type d'un processus = effets qu'il produit
- si a est de niveau 3 et b de niveau 4, $P = \overline{a}\langle t\rangle \mid\mid \overline{b}\langle t'\rangle$ a le type $\{3, 4\}$.
- type d'un canal = $Ch^{\text{niveau du canal}}(\text{type des données transportées})$
- typage d'un envoi :

$$\frac{\Gamma(a) = Ch^k(\tau) \qquad \Gamma \vdash t : \tau}{\Gamma \vdash \overline{a}\langle t\rangle : \{k\}}$$

- typage d'une réception :

$$\frac{\Gamma(a) = Ch^k(\tau) \qquad \Gamma, x : \tau \vdash P : e \qquad e < \{k\}}{\Gamma \vdash a(x).P : \emptyset}$$

Typage en action

- terme divergent : $S_0 \equiv a(k).k\ ()\ ||\overline{a}\langle k\rangle$
  impossible de dériver un jugement de typage (règle réception non appliquable)
- terme terminant (redirection $b \to c$) :

$$\texttt{let}\ S\ ()\ =\ a^1(k^{\texttt{unit}\to\emptyset}).(k\ ()\ ||\ b^2(x).(\overline{c}^1\langle x\rangle\ ||\ \overline{a}^1\langle k\rangle))\ \texttt{in}\ S\ ()\ ||\ \overline{a}^1\langle S\rangle$$

Typage en $\lambda\pi$ (règles)

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x.t : \tau \to \sigma}$$

$$\frac{\Gamma \vdash t : \tau \to \sigma \qquad \Gamma \vdash u : \tau}{\Gamma \vdash t\ u : \sigma} \qquad \frac{}{\Gamma \vdash 0 : \emptyset}$$

$$\frac{\Gamma \vdash t : e \qquad \Gamma \vdash u : e'}{\Gamma \vdash t \parallel u : e \uplus e'} \qquad \frac{\Gamma(a) = \mathsf{Ch}_k(\tau) \qquad \Gamma \vdash t : \tau}{\Gamma \vdash \overline{a}\langle t \rangle : \{k\}}$$

$$\frac{\Gamma(a) = \mathsf{Ch}_k(\tau) \qquad \Gamma, x : \tau \vdash t : e' \qquad e' < \{k\}}{\Gamma \vdash a(x).t : \emptyset}$$

$$\frac{\Gamma, a : \mathsf{Ch}_k(\tau) \vdash t : \tau}{\Gamma \vdash (\nu a)t : \tau} \qquad \frac{\Gamma \vdash t : \sigma \qquad \sigma \preceq \sigma'}{\Gamma \vdash t : \sigma'}$$

Aperçu de la preuve

Théorème

Si un terme t de $\lambda\pi$ est typable, alors il termine.

- preuve par candidats de réductibilité à la Girard
- pas de problèmes pour la partie séquentielle
- pour la partie concurrente, lors d'une réduction, diminution du poids d'un processus ;
- quand on a $a(x).P \mid\mid \overline{a}\langle t\rangle \rightarrow P[t/x]$, P contient que des envois sur des canaux plus petits que a
- $\Rightarrow$ décroissance pour l'ordre multi-ensembles
- ainsi : si $P||Q$ diverge, alors on fait forcément une communication à un moment qui diminuera les effets du processus et ainsi de suite.

Borne polynomiale et complexité implicite

Autre contribution du stage :

- Soft$\lambda\pi$: extension du langage et du système de type pour avoir une borne sur la longueur des réductions
- inspiré des idées de Soft Linear Logic (uniquement pour la partie séquentielle) (cf. *Soft linear logic and polynomial time*, Y. Lafont)
- résultat : pour un processus P, si
  - P utilise n canaux ;
  - dans P on ne dépasse pas k niveaux d'imbrication ;
  - aucune variable n'apparaît plus de d fois

  alors P se réduit vers sa forme normale en un nombre de réductions de l'ordre de $O(d^{n+k})$

Conclusion

- $\lambda\pi$: un langage pour étudier la terminaison des programmes mêlant concurrence et séquentialité
- intégration des *session types* ou des *i/o types*

SoftLambdaPi

$$\frac{\Gamma;\Delta_1 \vdash t : e \qquad \Gamma;\Delta_1 \vdash u : e'}{\Gamma;\Delta_1,\Delta_2 \vdash t \parallel u : e \uplus e'} \qquad \frac{\Gamma, a : \mathsf{Ch}_k(\tau);\Delta \vdash t : \sigma}{\Gamma;\Delta \vdash (\nu a)t : \sigma}$$

$$\frac{\Gamma;\Delta \vdash t : \tau \qquad \Gamma(a) = \mathsf{Ch}_k(\tau)}{\Gamma;\Delta \vdash \overline{a}\langle t\rangle : \{k\}} \qquad \frac{}{\Gamma;\emptyset \vdash 0 : \emptyset}$$

$$\frac{\Gamma, x : \tau;\Delta \vdash t : e \qquad \Gamma(a) = \mathsf{Ch}_k(\tau) \qquad e < \{k\}}{\Gamma;\Delta \vdash a(x).t : \emptyset}$$

$$\frac{\Gamma, x : \tau;\Delta \vdash t : \sigma}{\Gamma;\Delta \vdash \lambda!x.t : !\tau \to \sigma} \qquad \frac{\Gamma;\Delta, x : \tau \vdash t : \sigma}{\Gamma;\Delta \vdash \lambda x.t : \tau \to \sigma}$$

$$\frac{\Gamma;\Delta_1 \vdash t : \tau \to \sigma \qquad \Gamma;\Delta_2 \vdash u : \tau}{\Gamma;\Delta_1,\Delta_2 \vdash t\ u : \sigma} \qquad \frac{\Gamma \cup D(x) = \tau}{\Gamma;\Delta \vdash x : \tau}$$

$$\Gamma \vdash t : \tau \qquad \tau \leq \sigma$$

$\lambda\pi$ versus λ-calcul à régions (de R. Amadio)

- λ-calcul à régions : λ-calcul avec des régions mutables
- système de types pour ce langage avec terminaison via des candidats de réductibilité (plus compliqués)
- différences fondamentales :
  - λ-calcul à régions : appel par nom, aucune stratégie fixée en $\lambda\pi$;
  - la lecture d'une région est immédiate, l'écoute sur un canal est bloquante
    cela entraîne que le *spawning* n'a pas de sens
- $\lambda\pi$ accepte donc plus de gens en tenant compte des spécificités de π.