セッション型、簡潔に

オレッグ・キセリョーヴ
東北大学大学院情報科学研究科
oleg@okmij.org

今井 敬吾
岐阜大学工学部
https://keigoimai.info/

- セッション型付けに関する従来とは全く異なる実装アプローチ
- 既存セッション型システムを「魔術せずに」MetaOCaml 上で埋め込んだ
- 実装・エラーメッセージは誰でも理解できる
- 通常のデバッグ手法上で、セッション型エラーがデバッグできる
- 型検査と段階的計算との深い関係を注目させる

セッション型とは

通常 (式) 型	セッション型
$x : \mathsf{int} \vdash x + 1 : \mathsf{int}$	$\vdash x?[z]\ \mathsf{in}\ \ y![z{>}1]; \mathsf{finish} \triangleright x :?[\mathsf{int}]; \mathsf{end} \cdot y :![\mathsf{bool}]; \mathsf{end}$
式の評価結果の予想	通信進行の予想
静的に決る近似	
見完全プログラムにも与えられる	
HM っぽい	線形型っぽい
型付いたプログラムは don't go wrong:	
• int と string を足さない	• 意外なデータが届かない • チャネルで block しない • プロトコルに違反しない • 閉じた・委譲されたチャネルを操作しない

セッション型システム

既存の吉田・Vasconcelos (2007) による Liberal なセッション渡し型システムをそのまま採用した

例としたプロセス達

$$
\begin{aligned}
P_1 &: y_1?[z_1]\ \mathsf{in}\ \ y_2?[z_2]\ \mathsf{in}\ \ y_2![z_1{>}z_2]; \mathsf{finish} \\
P_2 &: \mathsf{accept}\, a(y_1)\ \mathsf{in}\ \ P_1 \\
P_3 &: \mathsf{request}\, b(y_2)\ \mathsf{in}\ \ P_2 \\
Q &: \mathsf{accept}\, b(x_2)\ \mathsf{in}\ \ \mathsf{request}\, a(x_1)\ \mathsf{in}\ \ x_1![1]; x_2![2]; x_2?[z]\ \mathsf{in}\ \ \mathsf{finish} \\
R &: P_3 \mid Q
\end{aligned}
$$

以下、解説しましょう

$$\Theta; \Gamma \vdash P \triangleright \Delta$$

$$\frac{\Gamma \vdash a : \langle \alpha, \bar{\alpha} \rangle \quad \Theta; \Gamma \vdash P \triangleright \Delta \cdot x : \alpha}{\Theta; \Gamma \vdash \mathsf{accept}\, a(x)\ \ \mathsf{in}\ \ P \triangleright \Delta}\ [Acc]$$

$$\frac{\Gamma \vdash e \triangleright S \quad \Theta; \Gamma \vdash P \triangleright \Delta \cdot y : \alpha}{\Theta; \Gamma \vdash y![e]; P \triangleright \Delta \cdot y :![S]; \alpha}\ [Send]$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot y : \beta}{\Theta; \Gamma \vdash \mathsf{throw}\, y[y']; P \triangleright \Delta \cdot y :![\alpha]; \beta \cdot y' {:} \alpha}\ [Thr]$$

例えば

$$\vdash P_1 \triangleright y_1 :?[\mathsf{int}]; \mathsf{end} \cdot y_2 :?[\mathsf{int}]; ![\mathsf{bool}]; \mathsf{end}$$

$$a : \langle ?[\mathsf{int}]; \mathsf{end}\, , ![\mathsf{int}]; \mathsf{end} \rangle \vdash P_2 \triangleright y_2 :?[\mathsf{int}]; ![\mathsf{bool}]; \mathsf{end}$$

P_3, Q, R のセッション型は自分で推論を試してみよう。デーモで確認しましょう。

もっと面白い例

$$
\begin{aligned}
&(\mathsf{accept}\, a(y)\ \mathsf{in}\ \ y![1]; y?[x]\ \mathsf{in}\ \ y![x{>}0]; y?[x']\ \mathsf{in}\ \ \mathsf{finish}) \mid \\
&(\mathsf{request}\, a(y)\ \mathsf{in}\ \ y?[x]\ \mathsf{in}\ \ y![x]; \mathsf{accept}\, b(z)\ \mathsf{in}\ \ \mathsf{throw}\, z[y]; \mathsf{finish}) \mid \\
&(\mathsf{request}\, b(z)\ \mathsf{in}\ \ \mathsf{catch}\, z[y]\ \mathsf{in}\ \ y?[x]\ \mathsf{in}\ \ y![\mathsf{not}\, x]; \mathsf{finish})
\end{aligned}
$$

また、算術サーバ

重点: 段階的計算

生成「された」コード立場	生成「する」コード (生成器) 立場
型検査	実行時検査
魅力的で高度な「型」	退屈な「項」

デモ

- MetaOCaml 上で表現
- セッション型推論、実行前にエラーを発見
- プロセスコード生成・実行

実装方法

独立 DSL

◯ (複雑でも) 任意の構文・型システムができる
╳独力でコンパイラを書くなんて厳しい
╳実装質を保証するのは難しい

埋め込んだ DSL

◯ ホスト言語の定義・高階関数・モジュルなど抽象機能、インフラなどを最利用
╳線形型等を持つ DSL を埋め込んだら、
– 極高度の型ススステム機能が要る
– 魔術ほどな能力・発想が必要
– その実装やエラーメッセージを理解するのは厳しい
– (C++ template metaprogramming は思い出させる)

埋め込んだ DSL ＋段階計算

◯ 埋め込んだ DSL の全ての利点
◯「ある程度に」実装質を静的に保証
– ホスト言語の型システム上で、素朴な誤りを防ぐ
– 少くとも、生成されたコードは必ず型付いたもの

GADT 上で段階計算の実装かなぁ　.<let x=Channel.receive ep in .~(k .<x>.)>. を試みて

http://okmij.org/ftp/meta-programming/sessions/