Derivation Rules in Fialyzer

fialyzer developers

This file shows derivation rules used in fialyzer.

Our derivation rules are almost same as the original success typings paper[*1]'s one, but extended by remote call, local call, list, etc.

1. Derivation Rules

Here are the BNFs used in the derivation rules:

$$
\begin{array}{lll}
e & ::= & v \mid x \mid fn \mid \{e,\cdots,e\} \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{letrec}\ x = fn,\cdots,x = fn\ \texttt{in}\ e \\
 & \mid & e(e,\cdots,e) \mid \texttt{case}\ e\ \texttt{of}\ pg \to e;\cdots;\ pg \to e\ \texttt{end} \mid \texttt{fun}\ f/a \mid \texttt{[}e\texttt{|}e\texttt{]} \mid \texttt{[]} \mid \texttt{fun}\ m\!:\!f/a \quad \text{(term)} \\
v & ::= & \texttt{0} \mid \texttt{'ok'} \mid \cdots \quad \text{(constant)} \\
x & ::= & \text{(snip)} \quad \text{(variable)} \\
fn & ::= & \texttt{fun}(x,\cdots,x) \to e \quad \text{(function)} \\
pg & ::= & p\ \texttt{when}\ g;\cdots;g \quad \text{(pattern with guard sequence)} \\
p & ::= & v \mid x \mid \{p,\cdots,p\} \mid \texttt{[}p\texttt{|}p\texttt{]} \mid \texttt{[]} \quad \text{(pattern)} \\
g & ::= & v \mid x \mid \{e,\cdots,e\} \mid \texttt{[}e\texttt{|}e\texttt{]} \mid \texttt{[]} \mid e(e,\cdots,e) \quad \text{(guard)} \\
m & ::= & e \quad \text{(module name. a term to be an atom)} \\
f & ::= & e \quad \text{(function name. a term to be an atom)} \\
a & ::= & e \quad \text{(arity. a term to be a non\_neg\_integer)} \\
\tau & ::= & \texttt{none()} \mid \texttt{any()} \mid \alpha \mid \{\tau,\cdots,\tau\} \mid (\tau,\cdots,\tau) \to \tau \mid \tau \cup \tau \\
 & \mid & \texttt{integer()} \mid \texttt{atom()} \mid 42 \mid \texttt{'ok'} \mid \cdots \quad \text{(type)} \\
\alpha,\beta & ::= & \text{(snip)} \quad \text{(type variable)} \\
C & ::= & (\tau \subseteq \tau) \mid (C \land \cdots \land C) \mid (C \lor \cdots \lor C) \quad \text{(constraint)} \\
A & ::= & A \cup A \mid \{x \mapsto \tau,\cdots,x \mapsto \tau\} \quad \text{(context. mapping of variable to type)}
\end{array}
$$

Here are the derivation rules:

$$\frac{}{A \cup \{x \mapsto \tau\} \vdash x : \tau, \emptyset}\text{(VAR)}$$

1 T. Lindahl and K. Sagonas. Practical Type Inference Based on Success Typings. In*Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 167–178. ACM, 2006.

$$\frac{A \vdash e_1 : \tau_1, C_1 \qquad \cdots \qquad A \vdash e_n : \tau_n, C_n}{A \vdash \{e_1, \cdots, e_n\} : \{\tau_1, \cdots, \tau_n\}, C_1 \wedge \cdots \wedge C_n}(\text{STRUCT})$$

$$\frac{A \vdash e_1 : \tau_1, C_1 \qquad A \cup \{x \mapsto \tau_1\} \vdash e_2 : \tau_2, C_2}{A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2, C_1 \wedge C_2}(\text{LET})$$

$$\frac{A' \vdash fn_1 : \tau_1, C_1 \cdots A' \vdash fn_n : \tau_n, C_n \qquad A' \vdash e : \tau, C \qquad \texttt{where}\ A' = A \cup \{x_i \mapsto \alpha_i\}}{A \vdash \texttt{letrec}\ x_1 = f_1, \cdots, x_n = f_n\ \texttt{in}\ e : \tau, C_1 \wedge \cdots \wedge C_n \wedge C \wedge (\tau'_1 = \tau_1) \wedge \cdots \wedge (\tau'_n = \tau_n)}(\text{LETREC})$$

$$\frac{A \cup \{x_1 \mapsto \alpha_1, \cdots, x_n \mapsto \alpha_n\} \vdash e : \tau, C}{A \vdash \texttt{fun}(x_1, \cdots, x_n) \to e : (\alpha_1, \cdots, \alpha_n) \to \tau, C}(\text{ABS})$$

$$\frac{A \vdash e : \tau, C \qquad A \vdash e_1 : \tau_1, C_1 \cdots A \vdash e_n : \tau_n, C_n}{A \vdash e(e_1, \cdots, e_n) : \beta, (\tau = (\alpha_1, \cdots, \alpha_n) \to \alpha) \wedge (\beta \subseteq \alpha) \wedge (\tau_1 \subseteq \alpha_1) \wedge \cdots \wedge (\tau_n \subseteq \alpha_n) \wedge C \wedge C_1 \wedge \cdots \wedge C_n}(\text{APP})$$

$$\frac{A \vdash p : \tau, C_p \qquad A \vdash g : \tau_g, C_g}{A \vdash p\ \texttt{when}\ g : \tau, (\tau_g \subseteq \texttt{boolean()}) \wedge C_p \wedge C_g}(\text{PAT})$$

$$\frac{A \vdash e : \tau, C_e \qquad A_i \vdash pg_i : \tau_{pg_i}, C_{pg_i} \qquad A_i \vdash b_i : \tau_{b_i}, C_{b_i} \qquad \texttt{where}\ A_i = A \cup \{v \mapsto \alpha_v \mid v \in Var(pg_i)\}}{A \vdash \texttt{case}\ e\ \texttt{of}\ pg_1 \to b_1;\ \cdots\ pg_n \to b_n \texttt{end} : \beta, C_e \wedge (C_1 \vee \cdots \vee C_n) \texttt{where}\ C_i = ((\beta = \tau_{b_i}) \wedge (\tau = \tau_{pg_i}) \wedge C_{pg_i} \wedge C_{b_i})}(\text{CASE})$$

$$\frac{}{A \cup \{\texttt{fun}\ f/a \mapsto \tau\} \vdash \texttt{fun}\ f/a : \tau, \emptyset}(\text{LOCALFUN})$$

$$\frac{A \vdash e_1 : \tau_1, C_1 \qquad A \vdash e_2 : \tau_2, C_2}{A \vdash \texttt{[}e_1 \mid e_2\texttt{]} : \texttt{list}(\alpha \mid \tau_1), \tau_2 = \texttt{list}(\alpha) \wedge C_1 \wedge C_2}(\text{LISTCONS})$$

$$\frac{}{A \vdash \texttt{[]} : \texttt{list(none())}, \emptyset}(\text{LISTNIL})$$

$$\frac{}{A \vdash \texttt{[]} : \texttt{list}(\alpha), \emptyset}(\text{PATLISTNIL})$$

$$\frac{}{A \cup \{\texttt{fun}\ m{:}f/a \mapsto \tau\} \vdash \texttt{fun}\ m{:}f/a : \tau, \emptyset}\text{if } m \text{ and } f \text{ is atom literal, } a \text{ is non\_neg\_integer literal } (\text{MFA})$$

$$\frac{A \vdash m : \tau_m, C_m \qquad A \vdash f : \tau_f, C_f \qquad A \vdash a : \tau_a, C_a}{A \vdash \texttt{fun}\ m{:}f/a : \beta, (\tau_m \subseteq \texttt{atom()}) \wedge (\tau_f \subseteq \texttt{atom()}) \wedge (\tau_a \subseteq \texttt{number()}) \wedge C_m \wedge C_f \wedge C_a}\text{if } \diamond\ (\text{MFAEXPR})$$

$\diamond$: neither m, f is atom literal nor a is non_neg_integer literal

1.1. Differences from the original paper

The differences from the derivation rules on the original paper are as follows.

- α, β, and τ are clearly distinguished. τ is a type, and α, β are type variables.

- LET is fixed: e_2, not e.
- ABS is modified: τ and constrained function are omitted.
- PAT is modified: type of g is `boolean()`, not `true`.
- CASE is fixed: τ, not τ_i. replaced $p_1 \cdots p_n$ with $pg_1 \cdots pg_n$ because these are patterns with guards.
- LOCALFUN is added.
- MFA is added.
- MFAEXPR is added.
- ...and some variables are α-converted for understandability.

1.2. Notes

- In $A \vdash p : \tau, C_p$ of PAT rule, p is not an expression but a pattern. Therefore, we have to convert p to an expression which is the same form of p.
    - This is not described in the original paper.