

Type Systems for Programming Languages (15-814)

Lecture Notes, Fall 2006, Week 9

Wenjie Fu

November 7 and 9, 2006

1 Computational Effects

So far, we discussed *effect-free model*, in which computation \sim calculation(evaluation).

Now, we are going to add effects. There are two kinds of effects.

1. Control Effects
non-local control transfer / context sensitive evaluation
2. Storage Effects
mutation of the environment
eg) I/O, assignable vars, mutable data.

2 Simple Exceptions / Failures

$$e ::= \dots (PCF) \dots \mid \mathbf{try} \ e_1 \ \mathbf{ow} \ e_2 \mid \mathbf{fail}$$
$$\frac{e_1:\tau \ e_2:\tau}{\mathbf{try} \ e_1 \ \mathbf{ow} \ e_2} \qquad \frac{}{\mathbf{fail}:\tau(\text{any type})}$$

eg) $\mathbf{fail}(3)$

2.1 Dynamic Semantics

1. Failure-passing –SOS
2. Control stack –AM(Abstract Machine)

2.2 Failure-passing

For any e , either $e \mapsto e'$ or e value ore fails.

We want $(\mathbf{fail}(e_2))(e_3)$ to be fails!

$$\begin{array}{c}
\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \qquad \frac{e_1 \text{ value} \quad e_2 \mapsto e'_2}{e_1 e_2 \mapsto e_1 e'_2} \qquad \frac{e_2 \text{ value}}{(\lambda x:\tau. e)e_2 \mapsto [e_2/x]e} \\
\\
\frac{}{\mathbf{fail} \text{ fails}} \qquad \frac{e_1 \text{ fails}}{e_1 e_2 \text{ fails}} \qquad \frac{e_1 \text{ value} \quad e_2 \text{ fails}}{e_1 e_2 \text{ fails}} \\
\\
\frac{e_1 \mapsto e'_1}{\mathbf{try } e_1 \text{ ow } e_2 \mapsto \mathbf{try } e'_1 \text{ ow } e_2} \qquad \frac{e_1 \text{ value}}{\mathbf{try } e_1 \text{ ow } e_2 \mapsto e_1} \qquad \frac{e_1 \text{ fails}}{\mathbf{try } e_1 \text{ ow } e_2 \mapsto e_2}
\end{array}$$

2.2.1 Safety

Preservation

If $e:\tau$ and $e \mapsto e'$, then $e':\tau$.

Progress

If $e:\tau$ then either $e \text{ value}$ or $e \text{ fails}$ or $\exists e'. e \mapsto e'$.

2.3 Abstract Machine

\mapsto make control state explicit.

We define *States* to be: (K is a *control stack*)

1. $K > e$ evaluation
2. $K < e$ return ($e \text{ value}$)
3. $K < \mathbf{fail}$ fail

$s_1 \mapsto s_2$ transition
 s final
 s initial

The definition of *Stack* is as follows.

$K ::= \epsilon \mid f; K$
 $f ::= \cdot(e_2) \mid e_1(\cdot) \mid \dots$

$\epsilon > e$ initial
 $\epsilon < e$ final (if $e \text{ value}$)
 $\epsilon < \mathbf{fail}$ final

$$\begin{aligned}
K > e_1 e_2 &\mapsto \cdot(e_2); K > e_1 \\
\cdot(e_2); K < e_1 &\mapsto e_1(\cdot); K > e_2 \text{ (where } e_1 \text{ value)} \\
(\lambda x:\tau. e)(\cdot); K < e_2 &\mapsto K > [e_2/x]e \text{ (where } e_2 \text{ value)} \\
K > \lambda x:\tau. e &\mapsto K < \lambda x:\tau. e \\
\cdot(e_2); K < \mathbf{fail} &\mapsto K < \mathbf{fail} \\
e_1(\cdot); K < \mathbf{fail} &\mapsto K < \mathbf{fail} \\
K > \mathbf{fail} &\mapsto K < \mathbf{fail} \\
K > \mathbf{try } e \mathbf{ ow } e_2 &\mapsto \mathbf{try } \cdot \mathbf{ ow } e_2; K > e_1 \\
\mathbf{try } \cdot \mathbf{ ow } e_2; K < e_1 &\mapsto K < e_1 \text{ (where } e_1 \text{ value)} \\
\mathbf{try } \cdot \mathbf{ ow } e_2; K < \mathbf{fail} &\mapsto K > e_2
\end{aligned}$$

How to RELATE **SOS** to **Abstract Machine**?

(harder) $e_1 \mapsto e_2$ only if ??? (for all K 's)

(easier) $K_1 > e_1 \mapsto K'_1 > e'_1$ only if ??? (reconstruction)

2.3.1 Safety for Abstract Machine

$$\left. \begin{array}{l} K > e \\ K < e \\ K < \mathbf{fail} \end{array} \right\} \text{ represent the entire program}$$

We want to define s ok.

Progress If s ok then either s final or $s \mapsto s' (\exists s')$

Preservation If s ok and $s \mapsto s'$, then s' ok

$$\frac{K \sim \tau \quad e:\tau}{K > e \text{ ok}}$$

($K \sim \tau$ represent K is a stack expecting a value of type τ .)

$$\overline{\epsilon \sim \tau}$$

(two ways of interpretation: (1)any type (2) τ_{ans} . we use (2) here)

$$\frac{\tau f\sigma; K \sim \sigma}{f; K \sim \tau}$$

(f takes τ and deliver σ)

eg)

$$\frac{e_2:\tau_2}{\tau_2 \rightarrow \tau \cdot (e_2) \ \tau} \qquad \frac{e_1:\tau_2 \rightarrow \tau}{\tau_2 \ e_1(\cdot) \ \tau} \qquad \frac{e_2:\tau}{\tau \ \mathbf{try} \cdot \mathbf{ow} \ e_2 \ \tau}$$

3 Exception

$e ::= \dots \mid \mathbf{try} \ e_1 \ \mathbf{ow} \ x.e_2 \mid \mathbf{throw} \ e$

Note: The **try** here is different from that in the previous section.

$$\frac{e_1:\tau \quad x :? \vdash e_2 : \tau}{\mathbf{try} \ e_1 \ \mathbf{ow} \ x.e_2 : \tau} \qquad \frac{e:?}", data-bbox="116 364 277 379" data-label="Text">

What should be “?”?$$

1. nat
correspond to “errno”
2. string x
The problem is how to handle it!
3. [div:1, fnf:string, ovf:1, ...] enumerate
We may write the handler as:

$$\mathbf{try} \ e_1 \ \mathbf{ow} \ x.\mathbf{case} \ x \ \{ \begin{array}{l} [\mathbf{div} = _] \Rightarrow \dots \\ [\mathbf{fnf} = s] \Rightarrow \dots \\ \dots \\ \} \end{array}$$

So, we want to define a τ_{exn} , but the problem is how to choose summands? Do we have to choose them at the very beginning? The answer is no.

4. extensible sum **exn**
(Note: It has nothing to do with exception at all. We’d better replace **exception** with **newtag** in **exception** z of τ)
The idea is to add a new summand to **exn**.

$$\mathbf{exn} = [A : \tau_A, B : \tau_B, \dots, Z : \tau_Z, \dots]$$

It looks like 3, but it always have $_ \Rightarrow \dots$ (be prepared for new summand) to support modularity.

3.1 Example

“Early exit” scenario

to cont(K)
 $L > \text{throw } F \text{ (letcc } x \text{ in throw } x \text{ to cont(K))}$
to cont(K)

$\underline{F(\cdot); \text{throw } \cdot \text{ to cont(K)}; L > \text{(letcc } x \text{ in throw } x \text{ to cont(L))}}$
 For convenience, let L' denotes the underline part

$L' > \text{throw cont(L')} \text{ to cont(L)}$
 $L < \text{cont(L')}$

3.5 Compiling Exceptions using Continuations

$\Gamma \vdash e : \tau$ PCF with exception

$\Gamma^+ \vdash e^+ : \chi \text{ cont} \rightarrow \tau^+$ PCF with cont

↑
 χ is the type of values carried by exceptions. In fact, τ_{exn} might be a better name for it.

3.5.1 Type Transform

$$\begin{aligned} \text{nat}^+ &:= \text{nat} \\ (\sigma \rightarrow \tau)^+ &:= \sigma^+ \rightarrow ((\chi \text{ cont}) \rightarrow \tau^+) \\ &\cong (\sigma^+ \times (\chi \text{ cont})) \rightarrow \tau^+ \\ \cdot^+ &:= \cdot \\ (\Gamma, x:\tau)^+ &:= \Gamma^+, x:\tau^+ \end{aligned}$$

3.5.2 Expression Transform

$$\begin{aligned} x^+ &:= \lambda h. x && \textit{h handler} \\ \text{zero}^+ &:= \lambda h. \text{zero} \\ \text{succ}(e)^+ &:= \lambda h. \text{succ}(e^+(h)) \\ \text{ifz}(e_0; e_1; x. e_2)^+ &:= \lambda h. \text{ifz}(e_0^+(h); e_1^+(h); x. e_2^+(h)) \\ e_1(e_2)^+ &:= \lambda h. e_1^+(h)(e_2^+(h))(h) \\ (\lambda x:\sigma. e)^+ &:= \lambda h. \lambda x:\sigma^+. \underline{\lambda h':\chi \text{ cont}. e^+(h')} && \text{underline part is } e^+ \end{aligned}$$

$$\begin{aligned}
\text{raise}(e)^+ &= \lambda h. \text{throw } e^+(h) \text{ to } h \\
(\text{try } e_1 \text{ ow } x.e_2)^+ &= \lambda h. \text{let } h' \text{ be} \\
&\quad \text{letcc } ret \text{ in} \\
&\quad \quad \text{let } x \text{ be} \\
&\quad \quad \quad \text{letcc } h' \text{ in throw } h' \text{ to } ret \\
&\quad \quad \quad \text{in } e_2^+(h) \\
&\quad \text{in } e_1^+(h')
\end{aligned}$$

Note: There's actually a mistake in the expression translation – the “try” case is not quite well-typed, a mistake in lecture. Thanks to William for pointing it out and giving the following correction and detailed explanation.

The problem is that in defining h' , we need to produce a χ cont. Therefore the body of the “letcc ret” expression needs to be a χ cont. Therefore the body of the “let x be” expression needs to be a χ cont, but it's not – it's the result of running $e_2^+(h)$.

Really, what we need to do is in the case of failure execution, throw the result to a context outside the definition of h' . So we need another letcc and a throw. So the last expression translation should be:

$$\begin{aligned}
(\text{try } e_1 \text{ ow } x.e_2)^+ &= \lambda h. \text{letcc } failure \text{ in} \\
&\quad \text{let } h' \text{ be} \\
&\quad \quad \text{letcc } ret \text{ in} \\
&\quad \quad \quad \text{let } x \text{ be} \\
&\quad \quad \quad \quad \text{letcc } h' \text{ in throw } h' \text{ to } ret \\
&\quad \quad \quad \quad \text{in throw } e_2^+(h) \text{ to } failure \\
&\quad \text{in } e_1^+(h')
\end{aligned}$$

What the above says is that:

- (1) We first take in the current handler h
- (2) We then grab the continuation at that point, in case of a failure exit
- (3) We next define a new handler continuation, h' , which should be the continuation that, when thrown to, binds the value thrown to x and runs e_2^+ with the original handler h
- (4) Finally, we run e_1^+ with that new handler