Control Flow

张昱

School of Computer Science and Technology
University of Science and Technology of China

Fall, 2018

控制流

1 用于控制的抽象机[PFPL, 28]

2 异常(Exception)[PFPL, 29]

3 延续(Continuations)[PFPL, 30]

1 用于控制的抽象机-1

把动态语义描述成转换系统

- 好处：对证明类型安全等理论目的十分有用。
- 不足：抽象级别太高，以至于不能在语言实现中直接使用。

  原因：用搜索规则时需要遍历和重构表达式，以简化其中的一小部分

  例如：加法的指令转换规则**(instruction transition rules)**

$$\frac{n_1 + n_2 = n \text{ nat}}{\texttt{plus}(\texttt{num}[n_1];\texttt{num}[n_2]) \mapsto \texttt{num}[n]}$$

  搜索转换规则**(search transition rules)**

$$\frac{e_1 \mapsto e_1'}{\texttt{plus}(e_1;e_2) \mapsto \texttt{plus}(e_1';e_2)} \qquad \frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{\texttt{plus}(e_1;e_2) \mapsto \texttt{plus}(e_1;e_2')}$$

  遍历：在**plus(e_1; e_2)**中搜索**e_1**；

  重构：将**plus(e_1; e_2)**重构为**plus(e'_1; e_2)**

1 用于控制的抽象机-2

在实现时，会更愿意用某种机制来记录“当前位于表达式哪儿”，这样在执行简化后可以从所记录的位置“恢复”。

——可以通过“控制栈”(跟踪记录指令步的上下文)来达到。

利用控制栈，转换规则无须任何前提，即每一规则都是公理！

在实现时，就无须对表达式进行遍历或重构。

真实机器和抽象机

- 在详细说明动态语义时，如果对实现所需的机制(如控制栈等)揭示得越多，则越接近真实机器。
- 顺着上述思路，从语言的结构语义开始，逐步引入实现所需的多种机制，向汇编级描述发展，所得到的描述即是特定的抽象机。
- 动态语义描述越接近真实机器，则它越“具体”，而越不“抽象”。
- 抽象和具体之间没有明显的分界线，而与实现细节被揭示的程度有关。

1 用于控制的抽象机-3

近年来的一个研究热点是用抽象机提供硬件无关的计算平台：
定义一个级别足够低的抽象机，使得：

- 在典型的硬件平台上易于实现；
- 较高级的语言可以翻译(编译)到这个抽象机

目标：大多数软件与具体的硬件平台无关。

对抽象机的要求：应被精确地定义。否则，不清楚如何翻译到抽象机，如何在给定的平台上实现该抽象机。

为L{nat ⇀}引入抽象机K{nat ⇀}，该抽象机引入控制栈来维护一个计算中尚未完成的子计算。

1.1 抽象机定义

1.2 安全性

1.3 控制机的正确性： L{nat ⇀}与K{nat ⇀}是等价的

Plotkin的PCF：L{nat $\rightharpoonup$}

❖ 语法

Types $\tau ::= \text{nat} \mid \tau_1 \rightharpoonup \tau_2$ $\rightharpoonup$ is partial function

Terms $t ::= x \mid \text{z} \mid s(t)$ variable, zero, and t's successor

$\mid \text{ifz}(t; t_0; x.t_1)$ if t is z, then t_0;

otherwise bind t' to x and compute t_1

where $t = s(t')$

$\mid \lambda x : \tau.t \mid t_1\ t_2$ lambda abstraction and application

$\mid \text{fix}(\lambda(x : \sigma).t)$ general recursion

Plotkin的PCF：L{nat →}

- 静态语义

变元的定型规则

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\,(\text{T-var})$$

nat 的引入规则(零)

$$\frac{}{\Gamma \vdash \mathrm{z} : \mathrm{nat}}\,(\text{T-z})$$

nat的引入规则(后继)

$$\frac{\Gamma \vdash t : \mathrm{nat}}{\Gamma \vdash s(t) : \mathrm{nat}}\,(\text{T-s})$$

$$\frac{\Gamma \vdash t : \mathrm{nat} \quad \Gamma \vdash t_0 : \tau \quad \Gamma, x : \mathrm{nat} \vdash t_1 : \tau}{\Gamma \vdash \mathrm{ifz}(t; t_0; x.t_1) : \tau}\,(\text{T-ifz})$$

条件分支的定型规则

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).t : \tau_1 \rightharpoonup \tau_2}\,(\text{T-pfn})$$

函数的引入规则

一般递归式的定型规则

$$\frac{\Gamma \vdash t_1 : \tau_2 \rightharpoonup \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\, t_2 : \tau}\,(\text{T-app})$$

函数的消去规则

$$\frac{\Gamma, x : \sigma \vdash t : \sigma}{\Gamma \vdash \mathrm{fix}(\lambda(x : \sigma).t) : \sigma}\,(\text{T-fix})$$

Plotkin的PCF：L{nat →}

❖ 动态语义

- •惰性(lazy)语义 省去{ }中的规则或前提
- •急切(eager)语义 包含{ }中的规则或前提

➢ Values

$$\frac{}{z\ \mathrm{val}}\ (\text{V-z}) \qquad \frac{\{t\ \mathrm{val}\}}{s(t)\ \mathrm{val}}\ (\text{V-s}) \qquad \frac{}{\lambda(x:\tau).t\ \mathrm{val}}\ (\text{V-fn})$$

➢ Dynamics

在eager语义下,t未完成求值,允许在后继下归约

$$\left\{\frac{t \mapsto t'}{s(t) \mapsto s(t')}\ (\text{D-s})\right\} \qquad \left\{\frac{t \mapsto t'}{\mathrm{fix}(t) \mapsto \mathrm{fix}(t')}\ (\text{D-fix}_1)\right\}$$

$$\frac{}{\mathrm{fix}(\lambda(x:\sigma).t) \mapsto [\mathrm{fix}(\lambda(x:\sigma).t)\ /\ x]t}\ (\text{D-fix}_2)$$

$$\frac{t_1 \mapsto t_1'}{t_1\ t_2 \mapsto t_1'\ t_2}\ (\text{D-app}_1) \qquad \frac{t \mapsto t'}{\mathrm{ifz}(t; t_0; x.t_1) \mapsto \mathrm{ifz}(t'; t_0; x.t_1)}\ (\text{D-ifz}_1)$$

$$\left\{\frac{t_1\ \mathrm{val} \quad t_2 \mapsto t_2'}{t_1 t_2 \mapsto t_1 t_2'}\ (\text{D-app}_e)\right\} \qquad \frac{}{\mathrm{ifz}(z; t_0; x.t_1) \mapsto t_0}\ (\text{D-ifz}_2)$$

$$\frac{\{t_2\ \mathrm{val}\}}{(\lambda(x:\tau).t_1)t_2 \mapsto [t_2\ /\ x]t_1}\ (\text{D-app}_2) \qquad \frac{}{\mathrm{ifz}(s(t); t_0; x.t_1) \mapsto [t\ /\ x]t_1}\ (\text{D-ifz}_3)$$

1.1 抽象机定义-1

- 抽象机K{nat ⇀}的状态
  - 状态 s 的组成：一个控制栈 k 和一个闭式 e
  - 状态的两种形式
    - 求值状态(evaluation state)：$k \triangleright e$
      该状态表示对与控制栈 k 相关的闭式 e 进行求值
    - 返回状态(return state)：$k \triangleleft e$，其中e val
      该状态表示与闭值e 相关的控制栈 k 进行求值

    状态中的分隔符 $\triangleright$ 或 $\triangleleft$ 指示该状态的焦点，即是求值状态中的表达式，还是返回状态中的栈
- 控制栈：表示求值上下文
  - 记录求值的“当前位置”及其上下文，上下文指出当前表达式所求得的值将被返回到哪儿。

1.1 抽象机定义-2

- 控制栈是一个栈帧(frame)列表，其定义如下

$$\frac{}{\varepsilon\ \text{stack}} \qquad \frac{f\ \text{frame} \quad k\ \text{stack}}{f;k\ \text{stack}}$$

- 栈帧的定义：依赖于当前求值所使用的语言
- K{nat ⇀}的栈帧由如下规则归纳定义
  与L{nat ⇀}的动态语义中带转换前提的各规则相对应

$$\frac{\left\{\dfrac{t \mapsto t'}{s(t) \mapsto s(t')}\right\}}{\text{ifz}(t;t_0;x.t_1) \mapsto \text{ifz}(t';t_0;x.t_1)} \qquad \frac{}{s(-)\ \text{frame}}$$

$$\frac{t \mapsto t'}{\text{ifz}(-;t_0;x.t_1)\ \text{frame}}$$

“—”表示当前的求值位置，s(—)是上下文

$$\frac{t_1 \mapsto t_1'}{t_1 t_2 \mapsto t_1' t_2} \qquad \frac{}{(-\ t_2)\ \text{frame}}$$

$$\frac{\{t_2\ \text{val}\}}{(\lambda(x:\tau).t_1)t_2 \mapsto [t_2\ /\ x]t_1} \qquad \frac{t_1\ \text{val}}{(t_1\ -)\ \text{frame}}$$

用控制栈上的帧来显式记录转换推导中尚未完成的计算!

1.1 抽象机定义-3

❖ K{nat ⇀ }的状态转换

➢ 自然数的转换规则

对z求值，则直接返回z

$$\frac{}{k \triangleright \mathrm{z} \mapsto k \triangleleft \mathrm{z}}\ (\text{K-z})$$

对s(t)求值，则将未完成的后继运算入栈，并对t求值

$$\frac{}{k \triangleright s(t) \mapsto s(-); k \triangleright t}\ (\text{K-s}_1)$$

当返回 t 给栈 时，则出栈并返回s(t) 给栈

$$\frac{}{s(-); k \triangleleft t \mapsto k \triangleleft s(t)}\ (\text{K-s}_2)$$

➢ 分情况分析的转换规则

将未完成的运算记录到栈中，并对t求值

$$\frac{}{k \triangleright \mathrm{ifz}(t; t_0; x.t_1) \mapsto \mathrm{ifz}(-; t_0; x.t_1); k \triangleright t}\ (\text{K-ifz}_1)$$

当返回 z 给栈时，则出栈并对t0求值

$$\frac{}{\mathrm{ifz}(-; t_0; x.t_1); k \triangleleft \mathrm{z} \mapsto k \triangleright t_0}\ (\text{K-ifz}_2)$$

当返回s(t) 给栈时，则出栈并用前驱t置换t1中出现的x

$$\frac{}{\mathrm{ifz}(-; t_0; x.t_1); k \triangleleft s(t) \mapsto k \triangleright [t\ /\ x]t_1}\ (\text{K-ifz}_3)$$

1.1 抽象机定义-4

❖ K{nat ⇀ }的状态转换

- 函数和递归式的转换规则

对λ 抽象求值，则直接返回之

$$\frac{}{k \triangleright \lambda(x:\tau).t \mapsto k \triangleleft \lambda(x:\tau).t}\ (\text{K-ptfn})$$

对λ 应用求值，则先计算被应用的 函数，即对**t1**求值

$$\frac{}{k \triangleright t_1\ t_2 \mapsto (-\ t_2); k \triangleright t_1}\ (\text{K-app}_1)$$

已得到被应用的函数(λ 抽象或递归函数)后，则对参数**t2**求值

$$\frac{}{(-\ t_2); k \triangleleft t_1 \mapsto (t_1\ -); k \triangleright t_2}\ (\text{K-app}_2)$$

t1是函数，**t2**是值时，则出栈并执行置换操作

$$\frac{t_1 = \lambda(x:\tau).t}{(t_1\ -); k \triangleleft t_2 \mapsto k \triangleright [t_2 / x]t}\ (\text{K-app}_3)$$

$$\text{fun}[\tau_1;\tau_2](x.y.t) = \text{fix}(\lambda(x:\tau_1 \rightharpoonup \tau_2).\lambda(y:\tau_1).t)$$

在当前栈**k**上，对一般递归式进行展开求值——无须栈空间

$$\frac{t_2\ \text{val} \quad t_1 = \lambda(x:\tau.t)}{(t_1\ -)\ ; k \triangleleft t_2 \mapsto k \triangleright [t_2 / x]t}$$

$$\frac{}{k \triangleright \text{fix}(\lambda(x:\tau).t) \mapsto k \triangleright [\text{fix}(\lambda(x:\tau).t) / x]t}\ (\text{K-fix})$$

1.1 抽象机定义-5

- K{nat ⇀}的初始状态和终结状态
  - 初始状态

$$\frac{}{\varepsilon \triangleright t \text{ initial}}\,(\text{K-init})$$

在空栈下对 t 求值

  - 终结状态

$$\frac{t \text{ val}}{\varepsilon \triangleleft t \text{ final}}\,(\text{K-final})$$

将值 t 返回给空栈

1.2 安全性-1

- 定型断言 $k : \tau$
  - k是良构的(well-formed)且期待类型为τ的值
  - 所谓栈是良构的，是指当向栈传递合适类型的值时，栈能安全地把这个值变换成一个解答(answer)，即程序的最终结果。
  - 类型τ_{ans}：表示求值结束的程序所具有的类型，它应是一个其值可直接观测的类型。

    对于L{nat ⇀}来说，τ_{ans}是nat.

  定型断言的含义：栈k 将一个类型为τ的值变换成一个类型为τ_{ans}的值
- 定型断言$k : \tau$由如下规则归纳定义：

$$\frac{}{\varepsilon : \tau_{ans}} \qquad \frac{k : \tau' \quad f : \tau \Rightarrow \tau'}{f;k : \tau} \qquad (29.7)$$

  - 辅助断言 $f : \tau => \tau'$表示栈帧f 将类型为τ的值变换成类型为τ'的值.

1.2 安全性-2

- K{nat ⇀ }中辅助断言 $f : \tau \Rightarrow \tau'$的归纳定义

$$\frac{}{s(-) : \text{nat} \Rightarrow \text{nat}}\,(\text{T-ks})$$

$$\frac{t_0 : \tau \quad x : \text{nat} \vdash t_1 : \tau}{\text{ifz}(-; t_0; x.t_1) : \text{nat} \Rightarrow \tau}\,(\text{T-kifz})$$

$$\frac{t_2 : \tau_2}{(-\ t_2) : \tau_2 \rightharpoonup \tau \Rightarrow \tau}\,(\text{T-kapp}_1)$$

$$\frac{t_1 : \tau_2 \rightharpoonup \tau \quad t_1\ \text{val}}{(t_1\ -) : \tau_2 \Rightarrow \tau}\,(\text{T-kapp}_2)$$

对栈帧ifz(－;t0;x.t1)来说，当前的求值位置”－”需要类型为nat的值， 该栈帧的结果类型为τ

- 以下规则定义了K{nat ⇀ }的状态是良构的

$$\frac{k : \tau \quad t : \tau}{k \triangleright t\ \text{ok}} \qquad \frac{k : \tau \quad t : \tau \quad t\ \text{val}}{k \triangleleft t\ \text{ok}}$$

- 安全性(PFPL)

1. 如果 s ok且 $s \mapsto s'$，则s' ok
2. 如果 s ok，则或者s final或者存在s'，使得 $s \mapsto s'$

1.3 控制机的正确性-1

- **K{nat ⇀}的正确性：是否正确实现L{nat ⇀}?**

  对该问题的回答可分解成如下两个命题

  - 完备性(completeness)

    如果$t \mapsto^* t'$ ，其中t' val，则 $\varepsilon \triangleright t \mapsto^* \varepsilon \triangleleft t'$

  - 可靠性(soundness)

    如果$\varepsilon \triangleright t \mapsto^* \varepsilon \triangleleft t'$，则 $t \mapsto^* t'$ ，其中t' val

- **完备性的证明**

  对多步转换定义进行归纳证明，则完备性定理简化为：

  1. 如果t val，则$\varepsilon \triangleright t \mapsto^* \varepsilon \triangleleft t$
  2. 如果$t \mapsto t'$，则对于每个v val，如果$\varepsilon \triangleright t' \mapsto^* \varepsilon \triangleleft v$，则 $\varepsilon \triangleright t \mapsto^* \varepsilon \triangleleft v$

1.3 控制机的正确性-2

❖ 完备性的证明

1.如果t **val**，则$\varepsilon \triangleright t \mapsto^{*} \varepsilon \triangleleft t$

证明：对t的结构进行归纳证明。

1) t是**z**，则由**(K-z)** 有$\varepsilon \triangleright \mathbf{z} \mapsto \varepsilon \triangleleft \mathbf{z}$

2) t是$s(t')$，则由**(K-s$_1$)**和**(K-s$_2$)**和归纳假设有

$\varepsilon \triangleright s(t') \mapsto^{*} \varepsilon \triangleleft s(t')$

3) t是$\lambda(x{:}\tau).t'$，则由**(K-ptfn)**可得。

$$\frac{}{k \triangleright \mathbf{z} \mapsto k \triangleleft \mathbf{z}}\,(\text{K-z})$$

$$\frac{}{k \triangleright s(t) \mapsto s(-); k \triangleright t}\,(\text{K-s}_1)$$

$$\frac{}{k \triangleright \lambda(x:\tau).t \mapsto k \triangleleft \lambda(x:\tau).t}\,(\text{K-ptfn})$$

$$\frac{}{s(-); k \triangleleft t \mapsto k \triangleleft s(t)}\,(\text{K-s}_2)$$

1.3 控制机的正确性-3

2.如果$t \mapsto t'$，则对于每个v val，如果$\varepsilon \triangleright t' \overset{*}{\mapsto} \varepsilon \triangleleft v$，则$\varepsilon \triangleright t \overset{*}{\mapsto} \varepsilon \triangleleft v$

对$t \mapsto t'$进行归纳分析，在证明中需要考虑如下两种复杂性

A.不能只考虑空栈。例如，若t为$(t_1\ t_2)$，则控制机的第一步为$\varepsilon \triangleright (t_1\ t_2) \mapsto (-\ t_2); \varepsilon \triangleright t_1$，这时就必须考虑在非空栈上对$t_1$求值。

B. 必须考虑每个子表达式的最终值。例如，若t为$(t_1\ t_2)$，t'为$(t'_1\ t_2)$且$t_1 \mapsto t'_1$。给定$k \triangleright (t'_1\ t_2) \mapsto^* k \triangleleft v$，则需要证明$k \triangleright (t_1\ t_2) \mapsto^* k \triangleleft v$。此证明推导的第一步是$k \triangleright (t'_1\ t_2) \mapsto (-\ t_2); k \triangleleft t'_1$，为应用归纳假设 $t_1 \mapsto t'_1$，必须有值v_1，使得$t'_1 \mapsto^* v_1$。

利用PFPL 7中的求值语义。

1.3 控制机的正确性-4

PFPL Lemma 28.2 如果 $e \Downarrow v$，则对每一 k stack，有 $k \triangleright e \mapsto^* k \triangleleft v$.

$e \Downarrow v$ e求值为 v

证明：对L{nat →}的求值语义进行归纳

考虑求值规则 $\dfrac{e_1 \Downarrow \lambda(x:\tau).e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1\ e_2 \Downarrow v}$

对任意控制栈k，需要证明$k \triangleright (e_1\ e_2) \mapsto^* k \triangleleft v$ 。

利用转换规则 (K-*) 来推导。

$$
\begin{aligned}
k \triangleright (e_1\ e_2) &\mapsto (-\ e_2); k \triangleright e_1 \\
&\mapsto^+ (-\ e_2); k \triangleleft \lambda(x:\tau_2).e \\
&\mapsto (\lambda(x:\tau_2).e\ -); k \triangleright e_2 \\
&\mapsto^+ (\lambda(x:\tau_2).e\ -); k \triangleleft v_2 \\
&\mapsto k \triangleright [v_2/x]e \\
&\mapsto^+ k \triangleleft v
\end{aligned}
$$

1.3 控制机的正确性-5

❖ 可靠性的证明

如果$\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft e'$，则$e \mapsto^* e'$，其中$e'$ val。

对多步转换$\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft v$的归纳推理比较麻烦，因为其间涉及求值状态和返回状态的交替。

取而代之，将每个机器状态当作对表达式的编码，从而可靠性证明就变成证明K{nat ⇀}的状态转换在这种编码下被L{nat ⇀}的状态转换所模拟。

断言$s \rightarrowtail e$：状态s 被分解为表达式项 e.

对于初始状态$s = \varepsilon \triangleright e$，或终结状态$s = \varepsilon \triangleleft e$，有$s \rightarrowtail e$.

则如果$s \mapsto^* s'$，其中s' final，$s \rightarrowtail e$且$s' \rightarrowtail e'$，
那么e' val且$e \mapsto^* e'$.

1.3 控制机的正确性-6

- 可靠性的证明

如果$s \mapsto^* s'$，其中s' final，$s \not\mapsto e$且$s' \not\mapsto e'$，则e' val且$e \mapsto^* e'$.

这反映如下两个事实：

1. 如果$s \not\mapsto e$且s final ，则e val。
2. 如果$s \mapsto s'$，$s \not\mapsto e$，$s' \not\mapsto e'$且$e' \mapsto^* v$，其中v val，则$e \mapsto^* v$.

针对1的证明，只需说明终结状态的分解式是一个值。

由(K-final),28.6b)终结状态的定义规则来得到。

针对2，需要证明下述引理

PFPL Lemma 28.3 如果$s \mapsto s'$，$s \not\mapsto e$且$s' \not\mapsto e'$，则$e \mapsto^* e'$.

1.3 控制机的正确性-7

❖ 可靠性的证明

- 断言 $s \looparrowright e$ 由辅助断言 $k \bowtie e = e'$ 按以下规则来定义，其中 s 是 $k \triangleright e$ 或 $k \triangleleft e$. (即，为分解状态而将栈打包在表达式上)

$$\frac{k \bowtie e = e'}{k \triangleright e \looparrowright e'} \qquad \frac{k \bowtie e = e'}{k \triangleleft e \looparrowright e'} \tag{28.11}$$

- 辅助断言 $k \bowtie e = e'$ 由以下规则定义

$$\frac{}{\epsilon \bowtie e = e}$$

$$\frac{k \bowtie \mathtt{s}(e) = e'}{\mathtt{s}(-);k \bowtie e = e'}$$

$$\frac{k \bowtie \mathtt{ifz}(e_1; e_2; x.e_3) = e'}{\mathtt{ifz}(-; e_2; x.e_3);k \bowtie e_1 = e'}$$

$$\frac{k \bowtie \mathtt{ap}(e_1; e_2) = e}{\mathtt{ap}(-; e_2);k \bowtie e_1 = e}$$

$$\frac{k \bowtie \mathtt{ap}(e_1; e_2) = e}{\mathtt{ap}(e_1; -);k \bowtie e_2 = e}$$

(28.12)

$(e_1\ e_2)$ can represent as $\mathtt{ap}(e_1; e_2)$

1.3 控制机的正确性-8

❖ 可靠性的证明

PFPL Lemma 28.4 断言$s \nrightarrow e$具有模式$(\forall, \exists!)$，
断言$k \bowtie e = e'$具有模式$(\forall, \forall, \exists!)$.

前者指每一状态将分解成1个唯一的表达式；后者指对打包有栈的表达式来说，其结果将唯一确定。从而对任意的k和e,存在e'，使得$k \bowtie e = e'$。

PFPL Lemma 28.5 如果$e \mapsto e'$，$k \bowtie e = d$，$k \bowtie e' = d'$，
则$d \mapsto d'$.

该引理说明状态分解将保持转换关系。

证明：对转换规则$e \mapsto e'$归纳证明。

- 基本步：转换规则是公理，这时对栈归纳分析来证明
- 归纳步：转换规则有前提，这时通过归纳来证明

1.3 控制机的正确性-9

- 可靠性的证明

PFPL Lemma 28.5 如果$e \mapsto e'$，$k \bowtie e = d$，$k \bowtie e' = d'$，则$d \mapsto d'$.

证明：对转换规则$e \mapsto e'$归纳证明。

- 归纳步：转换规则有前提，这时通过归纳来证明

假设e为$\mathrm{ap}(e_1; e_2)$，e'为$\mathrm{ap}(e'_1; e_2)$且$e_1 \mapsto e'_1$。

若$k \bowtie e = d$，$k \bowtie e' = d'$成立，

由规则(28.12)

$$\frac{k \triangleright\!\bowtie \mathrm{ap}(e_1; e_2) = e}{\mathrm{ap}(-; e_2); k \triangleright\!\bowtie e_1 = e}$$

则有$\mathrm{ap}(-; e_2); k \bowtie e_1 = d$，$\mathrm{ap}(-; e_2); k \bowtie e'_1 = d'$

由归纳假设，有$d \mapsto d'$.

1.3 控制机的正确性-10

PFPL Lemma 28.5 如果$e \mapsto e'$，$k \rhd\lhd e = d$，$k \rhd\lhd e' = d'$，则$d \mapsto d'$.

- 基本步：转换规则是公理，这时对栈归纳分析来证明

假设e为$\mathrm{ap}(\mathrm{lam}[\tau_2](x.e), e_2)$，$e'$为$[e_2/x]e$，则有$e \mapsto e'$。

若$k \rhd\lhd e = d$，$k \rhd\lhd e' = d'$成立，要证$d \mapsto d'$.

继续对k的结构进行归纳：

如果$k = \varepsilon$，则由 $\dfrac{}{\varepsilon \rhd\lhd e = e}$ 直接可得$d \mapsto d'$.

如果$k = \mathrm{ap}(-; c_2); k'$，则由 $\dfrac{k \rhd\lhd \mathrm{ap}(e_1; e_2) = e}{\mathrm{ap}(-; e_2); k \rhd\lhd e_1 = e}$

有$k' \rhd\lhd \mathrm{ap}(e, c_2) = d$，$k' \rhd\lhd \mathrm{ap}(e', c_2) = d'$

由结构语义规则，有$\mathrm{ap}(e; c_2) \mapsto \mathrm{ap}(e'; c_2)$，

再由内层的归纳假设，可得$d \mapsto d'$.

1.3 控制机的正确性-11

- 可靠性的证明

PFPL Lemma 28.3 如果$s \mapsto s'$,$s \not\mapsto e$且$s' \not\mapsto e'$，则$e \mapsto^* e'$.

证明：对K{nat ⇀}的转换分情况分析。针对每一情况，在对转换分解后，将对应L{nat ⇀}的0个或1个转换。

假设$s = k \triangleright \mathrm{s}(e)$且$s' = \mathrm{s}(-) \triangleright e$.

$k \bowtie \mathrm{s}(e) = e'$当且仅当$\mathrm{s}(-); k \bowtie e = e'$，由引理28.5有$e \mapsto e'$.

假设$s = \mathrm{ap}(\mathrm{lam}[\tau](x.e_1), -); k \triangleleft e_2$且$s' = k \triangleright [e_2/x]e_1$.

若有e'和e''，使得

$\mathrm{ap}(\mathrm{lam}[\tau](x.e_1), -); k \bowtie e_2 = e'$，$k \bowtie [e_2/x]e_1 = e''$.

有$k \bowtie \mathrm{ap}(\mathrm{lam}[\tau](x.e_1), e_2) = e'$. 由引理28.5有$e \mapsto^* e'$.

2 异常

异常：在引起异常处实现控制的非局部转移，动态地转移到处理该异常的外层处理器中。

控制转移将中断程序的正常控制流，以响应特殊条件。

例如，异常可用于发出错误条件信号或指示在某例外环境下所需的特殊条件。

通过异常可以避免对错误或特殊条件的一系列显式检查。

2.1 失败(Failures)

2.2 异常(Exceptions)

2.1 失败(Failures)-1

- 失败：一种简单的控制机制

  当求值失败时，将控制传给外层最近的处理器(捕获失败)

  - 失败：异常的一种简化形式，它没有关联的值
    ——将焦点放在对控制流的处理上

- 语法：对L{→}的扩展

  Expr's $e ::= \text{fail}[\tau] \mid \text{catch}(e_1; e_2)$

  - **fail[τ]**：中止当前的求值(引起一个失败)
  - **catch(e_1; e_2)**：对e_1求值，如果正常结束，则返回其值；
    如果失败，则返回e_2(失败处理器)的值.

  回顾：空和类型**void**的消去形式**abort[τ](e)**。

2.1 失败(Failures)-2

❖静态语义

$$\frac{}{\Gamma \vdash \mathtt{fail}[\tau] : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{catch}(e_1; e_2) : \tau} \qquad (30.1)$$

- 失败可以是任意类型，因为它从不会返回到失败处.
- 失败处理器**catch**的两个子句e_1和e_2必须是相同类型.

❖动态语义：应用栈展开(stack unwinding)技术

- 当对**catch**求值时，会在控制栈中安置一个失败处理器
- 当对**fail**求值时，会出栈直至遇到最近的外层失败处理器来展开控制栈，将控制传给该处理器
- 在当前控制栈上下文下对处理器求值，故处理器中的失败可以进一步向上传播

2.1 失败(Failures)-3

- 动态语义：应用栈展开(stack unwinding)技术

  可以用K{nat ⇀}来描述失败的动态语义

  - 引入新状态：$k \blacktriangleleft$

    该状态将失败传给栈，以便在栈中搜索最近的外层失败处理器

  - $\varepsilon \blacktriangleleft$：是final终结状态，不是受阻状态，对应于一个未捕获的失败("uncaught failure")

- 动态语义的定义

  - 栈帧的扩展

$$\frac{e_2 \ \text{exp}}{\text{catch}(-;e_2)\ \text{frame}} \qquad (30.2)$$

  栈帧：失败处理器

2.1 失败(Failures)-4

- 动态语义定义
  - 转换规则的扩展

对fail 求值，则传播一个失败给栈

对catch求值，则将失败处理器入栈，并对e1求值

如果传播给栈顶处理器一个值v(未失败)，则处理器出栈,并继续向栈传播v

如果传播给栈顶处理器一个失败，则处理器出栈,并对所存储的表达式e2进行求值

如果传播一个失败给栈，栈顶不是处理器，则出栈并继续向栈传播这个失败

$$\frac{}{k \triangleright \mathtt{fail}[\tau] \mapsto k \blacktriangleleft} \qquad (30.3)$$

$$\frac{}{k \triangleright \mathtt{catch}(e_1;e_2) \mapsto \mathtt{catch}(-;e_2);k \triangleright e_1}$$

$$\frac{}{\mathtt{catch}(-;e_2);k \triangleleft v \mapsto k \triangleleft v}$$

$$\frac{}{\mathtt{catch}(-;e_2);k \blacktriangleleft \mapsto k \triangleright e_2}$$

$$\frac{(f \neq \mathtt{catch}(-;e_2))}{f;k \blacktriangleleft \mapsto k \blacktriangleleft}$$

2.1 失败(Failures)-5

- 动态语义定义

$$\frac{e\ \text{val}}{\epsilon \lhd e\ \text{final}}$$

  - 终结状态的扩展

$$\frac{}{\epsilon \blacktriangleleft\ \text{final}}$$

    新增一个未捕获的异常，它可以在整个程序中传播

- 安全性(PFPL Theorem 29.1)

  1.如果 s ok且 $s \mapsto s'$，则s' ok .

  2.如果 s ok，则或者s final或者存在s'，使得 $s \mapsto s'$.

  注意：对进展性而言，这里的良类型程序不会受阻，但是可能会导致一个未被捕获的失败.

2.2 异常(Exceptions)-1

❖ 异常机制

当求值失败时，有关联的值，这个值将作为控制转移的一部分一起传给异常处理器(捕获异常)

❖ 语法

Expr's $e ::= \text{raise}[\tau](e) \mid \text{handle}(e_1; x.e_2)$

- **raise**$[\tau](e)$：对参数e求值，以确定传给处理器的值
- **handle**$(e_1; x.e_2)$：在对e_1求值期间，如果发生异常，则将异常所关联的值绑定到处理器e_2的约束变元x 上.

空和类型**void**的消去形式**abort**$[\tau](e)$ 。

2.2 异常(Exceptions)-2

$$\text{fun}[\tau_1;\tau_2](x.y.t) = \text{fix}(\lambda(x:\tau_1 \to \tau_2).\lambda(y:\tau_1).t)$$

❖ 异常举例:求阶乘函数

let $f = \lambda(n:\text{nat}).(\text{fun}[\text{nat};\text{nat}]f.u.\text{ifz}(u;s(\text{z});x.[(f\ x)\ /\ y]\text{times}\ (s(x),y))\ n)$

in $(f\ 3)$　　结果为6，求值过程相当于(3*(2*(1*(1))))

执行过程：逐层递归调用，然后逐层回归。

可以利用异常机制重新定义求阶乘函数！

let $f = \lambda(n:\text{nat}).(\lambda(r:\text{nat}).(\text{fun}[\text{nat};\text{nat}]f.u.$

$\text{ifz}(u;\text{raise}[\text{nat}](r);x.((f\ x)\ \text{times}\ (r,s(x))))\ n))$

$s(x)$

in handle($(f\ 3\ s(\text{z}))$; $ex.ex$)

f 的类型为nat→nat→nat,第2个参数r 将绑定到当前累乘的结果。

处理器捕获到异常值((1*3)*2)*1时,直接将其返回($ex.ex$).

结果为6，求值过程相当于((1*3)*2)*1

执行过程：这是一个迭代求值的过程。

2.2 异常(Exceptions)-3

❖动态语义

- 异常状态： $k \blacktriangleleft e$，其中e val.
- 栈帧的扩展

$$\frac{}{\texttt{raise}[\tau](-)\ \text{frame}}$$

$$\frac{}{\texttt{handle}(-;x.e_2)\ \text{frame}} \qquad (30.5)$$

记录对异常所关联的值求值时的上下文

- 对异常求值的转换规则

对raise求值，则将异常的求值上下文入栈，并对异常e求值

$$\frac{}{k \triangleright \texttt{raise}[\tau](e) \mapsto \texttt{raise}[\tau](-);k \triangleright e}$$

$$\frac{}{\texttt{raise}[\tau](-);k \triangleleft e \mapsto k \blacktriangleleft e}$$

$$\frac{}{\texttt{raise}[\tau](-);k \blacktriangleleft e \mapsto k \blacktriangleleft e} \qquad (30.6)$$

将异常状态传播给栈

2.2 异常(Exceptions)-4

❖动态语义

➢对异常求值的转换规则

对handle求值，则处理器入栈并对e1求值

未发生异常，则将处理器出栈，将值e继续传播给栈

$$\frac{}{k \triangleright \texttt{handle}(e_1; x.e_2) \mapsto \texttt{handle}(-; x.e_2); k \triangleright e_1}$$

$$\frac{}{\texttt{handle}(-; x.e_2); k \triangleleft e \mapsto k \triangleleft e} \quad (30.6)$$

发生异常，则将处理器出栈并用异常值e置换e2中的约束变元x，继续求值

$$\frac{}{\texttt{handle}(-; x.e_2); k \blacktriangleleft e \mapsto k \triangleright [e/x]e_2}$$

$$\frac{(f \neq \texttt{handle}(-; x.e_2))}{f; k \blacktriangleleft e \mapsto k \blacktriangleleft e}$$

若栈顶不是异常处理器，则出栈并继续将异常状态传播给栈

2.2 异常(Exceptions)-5

❖静态语义

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise}[\tau](e) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{handle}(e_1; x.e_2) : \tau} \qquad (30.7)$$

- τ_{exn}：表示异常类型

τ_{exn}应该是什么类型呢？

- 所有异常都应该类型相同，否则不能保证类型安全
  - 一个处理器可能会被任意在它保护的表达式执行期间所发生的 **raise**表达式调用
  - 若类型不同，则处理器不能静态预测异常值类型，从而不能在不违反类型安全的前提下分派处理异常。

2.2 异常(Exceptions)-6

τ_{exn}应该是什么类型呢？

- 将τ_{exn}看成是字符串：用来指示失败的原因
  - 如： **raise "Division by zero error."**
    raise "File not found."
  - 不足之处：对于那些会被异常处理器捕获的异常，处理器需要根据约定分析异常串，以确定发生了什么以及如何响应？
- 将τ_{exn}看成是**nat**：指示错误号
  - 不足之处：需要处理器根据错误号进行分派处理，这容易出错，因为数值不能体现关于异常的更多含义
- 将τ_{exn}看成是变式(带标签的和)类型：较为合理
  - 如： τ_{exn} **= [div:unit, fnf:string, …]**
    标签为**div**的项表示除零异常；标签为**fnf**的项表示文件未找到

2.2 异常(Exceptions)-7

τ_{exn}应该是什么类型呢？

- 将τ_{exn}看成是和类型：较为合理
  - 如： τ_{exn} = [div:unit, fnf:string, …]
  - handle处理器会根据变式的标签分情况分析，从而恢复成相应类型的数值（try e_1 ow $x.e_2$是handle(e_1; e_2)的具体语法）

```
try e1 ow x.case x {
  div <> => e_div
| fnf s => e_fnf
| … }
```

  - 不足之处：比较复杂

1) 必须以每个语言为基础来指定τ_{exn}

2) 选取单一固定的变式类型作为所有程序的τ_{exn}，这有些荒谬！除了一些基本的异常种类外，异常应该由程序员来选定。

2.2 异常(Exceptions)-8

如何为语言选择τ_{exn}，又能让程序员声明自己的异常？

- 将τ_{exn}定义成可扩展的和类型：[PFPL, 38]
  - 可扩展的和类型：允许动态创建新的标签，使得这种类型的可能的标签集合只能在运行时确定。

3 延续(Continuations)-1

控制栈：保存尚未完成的计算

许多控制结构(如异常、协程等)的语义可以根据一些具体控制栈来表达。

具体(reified)控制栈：将一个控制栈表示为一个普通的值，称为first-class continuation(延续)。
延续表示在给定时刻下，尚未完成的计算。

- 允许栈作为值在程序中传递，并在随后的某一地方能将当前控制栈恢复成过去的具体控制栈(即延续)。
- 所谓“first-class”是指这个值具有无限的生命期，可以随意地在程序中传递和返回。
  first-class continuation支持unlimited “time travel”(可以随意地回到从前，也可以返回到其未来的某一点)

如何实现具体控制栈，即延续？

- 将控制栈处理为持久的(persistent)数据结构（如保存在堆中）
  ——控制栈的新版本不会扰乱旧版本

3 延续(Continuations)-2

控制栈的标准实现：生命短暂的(ephemeral)数据结构

——不可能同时维护控制栈的“旧”、“新”副本，也不可能进行时间旅行！

First-class continuations的用处

- 反映一个计算在给定时刻的控制状态(尚未完成的计算)
- 利用continuations，可以”checkpoint”程序的控制状态，将其保存在数据结构中，以后再回到此控制状态

3.1 Continuation的简介

3.2 Continuation的语义

3.3 用Continuation表示异常

3.1 Continuation的简介-1

❖ 控制栈和Continuation(延续)

➢ 函数调用的实现

- 将现场(如当前指令的地址、一些寄存器的内容) 存入控制栈
- 传递实参，跳转到被调函数的代码处
- 传递被调函数的返回值并恢复现场

Continuation:在函数调用返回后需要继续进行的程序执行

➢ 示例1：

```
int f () { return 1; }
void main () {
  if (f()) printf("Good f\n");
  else printf("Bad f\n");
}
```

f( )的延续是：期待一个值v, 然后使用v继续执行直至程序结束。

3.1 Continuation的简介-2

```
int f () { return 1; }
void main () {
  if (f()) printf("Good f\n");
  else printf("Bad f\n");
}

void f_continuation (int v) {
  if (v) printf("Good f\n");
  else printf("Bad f\n");
}
```

f( )的延续是：期待一个值v, 然后使用v继续执行直至程序结束。

main变为

```
void main () {
  f_continuation(f());
}
```

- 将现场保存在f_continuation代码中
- 传递实参，跳转到被调函数f 的代码处
- 传递函数f 的返回值,作为f_continuation的实参，并调用f_continuation

3.1 Continuation的简介-3

```
int f () { return 1; }

void f_continuation (int v) {
  if (v) printf("Good f\n");
  else printf("Bad f\n");
}

void main () {
  f_continuation(f());
}
```

让 f 直接调用f_continuation;
从而归约成尾调用(tail call)

```
void f_continuation (int v) {
  if (v) printf("Good f\n");
  else printf("Bad f\n");
}

void f () { f_continuation(1); }

void main () { f(); }
```

尾调用可以优化成一个跳转。

3.1 Continuation的简介-4

❖控制栈和Continuation(延续)

➢示例2：求阶乘

```
int factorial (int n) {
  if (n == 0) return 1;
  else {
    int v = factorial(n-1);
    return n*v;
  }
}
void main () {
  printf ("Factorial of 5 = %d\n", factorial(5));
}
```

factorial(3)的延续是：

1) 期待一个值**v**

2) 用**v**乘以**4**，将结果返回给未完成的递归调用**factorial(5)**

3) 该递归调用将结果乘以**5**，产生最终结果

4) 将最终结果返回到最初的**factorial**调用，并打印

3.1 Continuation的简介-5

```
int factorial (int i, int prod) {
  if (i == 0) return prod;
  else return factorial(i-1, prod*i);
}
```

与2.2 异常-2 类似！
引入新的参数**prod**，保存当前累乘的结果。

尾调用优化

```
int factorial (int i, int prod) {
 L: {
    if (i == 0) return prod;
    else {
      prod = prod * i; i = i - 1; goto L;
    }
 }
}
```

这种程序风格称为是
CPS(Continuation-passing style)

3.1 Continuation的简介-6

❖ Continuation的类型与操作

- 类型：cont(τ)
  接受τ类型的值的Continuation类型
- 引入形式：letcc[τ](x.e)
  将当前的Continuation(当前控制栈)绑定到变量x，对e求值
  注意：letcc即为let with current continuation;
  还可以用函数表示cont(τ)的引入，即callcc(call with current continuation)
- 消去形式：throw(e_1; e_2)
  将控制栈恢复为e_2的值，再对e_1求值

抽象语法	具体语法
cont(τ)	τ cont
letcc[τ](x.e)	letcc x:τ cont in e
throw(e_1; e_2)	throw e_1 to e_2

3.1 Continuation的简介-7

❖ setjmp/longjmp与continuation

- setjmp/longjmp：是C的库函数，提供类似于异常处理的机制

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf k;

int f (int i) {
  if (i == 0) longjmp(k, 42);
  else return i*f(i-1);
}

void main () {
  int result = setjmp(k);
  switch (result) {
  case 0: f(5); break; // setjmp returns 0 first time
  default: printf("f(5) = %d\n", result); break; // w
  }
}
```

jmp_buf类型用来描述setjmp/longjmp所保护和恢复的现场信息，包括下一条指令地址、栈指针、一些寄存器的值等

跳回到k中保存的现场，将控制转移回setjmp(k)，此时返回42。longjmp在控制栈比调用setjmp处时的栈要大一些时才有意义，否则会导致悬空的栈指针！

将现场保存到k，并返回0。这个现场并不是当前的continuation!

程序执行的结果：42

3.1 Continuation的简介-8

❖ Continuation(延续)与求值上下文

- 上下文语义
  - 动态语义：由指令转换断言和求值上下文断言等进行定义
  - 项分解为当前的指令，和当前的**Continuation**(用求值上下文来形式化)
- 示例：

$$
\begin{aligned}
&\texttt{let}(\texttt{plus}(\texttt{num}[3],\texttt{num}[3]),x_1.\texttt{let}(\texttt{num}[4],x_2.\texttt{plus}(x_1,x_2))) \\
&\quad = \quad \texttt{let}(\circ,x_1.\texttt{let}(\texttt{num}[4],x_2.\texttt{plus}(x_1,x_2))\{\texttt{plus}(\texttt{num}[3],\texttt{num}[3])\} \\
&\quad \mapsto \quad \texttt{let}(\circ,x_1.\texttt{let}(\texttt{num}[4],x_2.\texttt{plus}(x_1,x_2))\{\texttt{num}[6]\} \\
&\quad = \quad \circ\{\texttt{let}(\texttt{num}[6],x_1.\texttt{let}(\texttt{num}[4],x_2.\texttt{plus}(x_1,x_2))\} \\
&\quad \mapsto \quad \circ\{\texttt{let}(\texttt{num}[4],x_2.\texttt{plus}(\texttt{num}[6],x_2))\} \\
&\quad \mapsto \quad \circ\{\texttt{plus}(\texttt{num}[6],\texttt{num}[4])\} \\
&\quad \mapsto \quad \circ\{\texttt{num}[10]\}
\end{aligned}
$$

当前的指令为 **plus(num[3];…)**
当前的延续为 **letcc [nat] (x. let(x;x1.let(…)))**

3.1 Continuation的简介-9

❖ Continuation的应用举例

示例1：自然数的无穷序列*q*的前*n*个元素的相乘

短路计算：如果在前*n*个元素中出现零，则希望尽早返回值0,而不是执行剩余的乘法。

➢ 无短路计算的解决方案

参数为无穷序列的索引，返回值为序列中对应的元素

```
fun ms (q : nat -> nat, n : nat) is
  case n {
    zero ⇒ succ(zero) |
    succ (m:nat) ⇒ times (q zero) (ms (q ∘ succ, m)) }
```

复合后得到一个新的序列，它比q少第0个元素；通过这个运算可以取得q中下一个元素！

3.1 Continuation的简介-10

示例1：自然数的无穷序列*q*的前*n*个元素的相乘

- 带短路计算的解决方案

```
λ (q : nat -> nat, n : nat) in
  letcc ret : nat cont in
    let fun ms (q : nat -> nat, n : nat) be
      case n {
          zero ⇒ succ(zero) |
          succ (m:nat) ⇒
            case (q zero) {
              zero => throw zero to ret |
              succ(p:nat) => times (succ p) (ms (q ∘ succ) m) } }
    in
      ms q n
```

ret将绑定到最初调用该函数时的continuation!

q的第0个元素为0，则将控制转移到ret，使得能尽早返回0！

3.1 Continuation的简介-11

示例2：(函数和延续的复合)给定类型为τ cont的延续k和类型为$\tau' \to \tau$ 的函数f，f和k复合后得到类型为τ' cont的延续k' 。

其行为如下：把类型为τ'的值v' 抛给k' 就是把值$f(v')$抛给k .

```
fun compose(f:τ' → τ,k:τ cont):τ' cont = ....
```

实现中的问题：1. 如何获得k'？ 2. 如何返回k'？

```
fun compose (f:τ' → τ, k:τ cont):τ' cont =
    letcc ret:τ' cont cont in
      throw (f (letcc r in throw r to ret)) to k
```

给定v',把 f(v')抛给k，这就相当于把v'抛给k'
——确定了要返回的延续

r类型为 τ ' cont; letcc的类型为τ ' r绑定到要返回的延续，throw r表示直接将传入的 r 抛给ret

3.2 Continuation的语义-1

- 对L{→}扩展以下内容

  Type　　$\tau ::= \text{cont}(\tau)$

  Expr　　$e ::= \text{letcc}[\tau](x.e) \mid \text{throw}(e_1; e_2) \mid \text{cont}(k)$

  - cont(*k*)：一个具体控制栈，它只在求值过程中出现，程序员不可以使用！

- 静态语义

$$\frac{\Gamma, x : \text{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \text{letcc}[\tau](x.e) : \tau}$$

throw表达式的类型是任意的，因为它不会回到调用它的地点

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \text{cont}(\tau_1)}{\Gamma \vdash \text{throw}[\tau'](e_1; e_2) : \tau'}$$

cont(k)是一个continuation，它接受类型为 τ 的值

$$\frac{k : \tau}{\Gamma \vdash \text{cont}(k) : \text{cont}(\tau)}$$

3.2 Continuation的语义-2

❖动态语义

- 扩展K{nat ⇀}的栈帧

$$\frac{e_2 \text{ exp}}{\texttt{throw}[\tau]\,(-;e_2)\text{ frame}} \qquad \frac{e_1 \text{ val}}{\texttt{throw}[\tau]\,(e_1;-)\text{ frame}}$$

- 每个具体控制栈(即continuation)是值

$$\frac{k \text{ stack}}{\texttt{cont}(k)\text{ val}}$$

- 转换规则

$$\frac{}{k \triangleright \texttt{letcc}[\tau]\,(x.e) \mapsto k \triangleright [\texttt{cont}(k)/x]e}$$

$$\frac{}{\texttt{throw}[\tau]\,(v;-)\,;k \triangleleft \texttt{cont}(k') \mapsto k' \triangleleft v}$$

$$\frac{}{k \triangleright \texttt{throw}[\tau]\,(e_1;e_2) \mapsto \texttt{throw}[\tau]\,(-;e_2)\,;k \triangleright e_1}$$

$$\frac{e_1 \text{ val}}{\texttt{throw}[\tau]\,(-;e_2)\,;k \triangleleft e_1 \mapsto \texttt{throw}[\tau]\,(e_1;-)\,;k \triangleright e_2}$$

3.2 Continuation的语义-3

❖ 安全性

- 增加以下栈帧的定型规则

$$\frac{e_2 : \mathtt{cont}(\tau)}{\mathtt{throw}[\tau](-; e_2) : \tau \Rightarrow \tau'}$$

$$\frac{e_1 : \tau \quad e_1 \ \mathsf{val}}{\mathtt{throw}[\tau](e_1; -) : \mathtt{cont}(\tau) \Rightarrow \tau'}$$

- 范式引理：如果**e : cont(τ)**且**e val**,则存在一个**k, e = cont(k)**使得**$k : \tau$** .
- 安全性定理

1.如果 **s ok**且 **$s \mapsto s'$**，则**s' ok** .

2.如果 **s ok**，则或者**s final**或者存在**s'**，使得 **$s \mapsto s'$**.

3.3 用Continuation表示异常-1

- 用延续表示异常的基本思想
  - 将当前的异常处理器表示成类型为τ_{exn} cont的延续,它是传到每个表达式e的隐含参数.从而有
    $e:\tau$ 翻译为$e':\tau_{exn}\ cont \rightarrow \tau$
  - 为引起一个异常raise[τ](e),会将异常值e 抛给当前的处理器
    raise[τ](e)翻译为lam[τ_{exn} cont](h.throw(e; h))
    h为当前的处理器，其类型为接受类型为τ_{exn}的延续类型
  - 为安装一个处理器handle(e_1; $x.e_2$),必须创建一个延续作为新的异常处理器,然后将这个新处理器传递给受保护的程序e_1

handle(e_1, $x.e_2$)(try e_1 ow $x.e_2$)翻译为类型为$\tau_{exn}\ cont \rightarrow \tau$的函数：

lam[τ_{exn} cont](h.letcc(r.let(…; h'.ap(e_1;h'))))

当handle应用到当前的异常处理器h时，上述λ函数将r 绑定到当前的延续，将h' 绑定到新产生的异常处理器，然后求ap(e_1;h')

3.3 用Continuation表示异常-2

handle(e_1, $x.e_2$)(try e_1 ow $x.e_2$)翻译为类型为τ_{exn} cont $\rightarrow \tau$的函数：

lam[τ_{exn} cont](h.letcc(r.let(…; h'.ap(e_1;h'))))

- 当handle应用到当前的异常处理器h时，上述λ函数将r 绑定到当前的延续，将h' 绑定到新产生的异常处理器，然后求ap(e_1;h')
- 绑定到h'的延续将控制转移到e_2，其中x绑定到e_1所产生的异常值.
- ap(e_1; h')确保如果e_1产生异常，则异常值可以被传递到h' .

h' 绑定到 letcc(r'.let(letcc(h'.throw(h'; r')); x.throw(ap(e_2; h),r))

- 设置一个内部的返回延续r'，然后建立所需的延续h' ，将h'抛给r'
- h'是什么呢？
  当产生异常时，将异常值绑定到x，用外层处理器h对e_2求值，将值抛给外层的返回延续r

3.3 用Continuation表示异常-3

小结：

当执行到任何一个表达式***e*** 时，都有一个当前的处理器***h*** 和当前未完成的计算(即(返回)延续)***r***。

当执行的表达式为**handle**时，在执行期间可能会发生异常，若发生异常，则要创建一个新的处理器***h'***，以及返回延续***r'***.

利用***h'*** 可以将产生的异常值绑定到***x***，再将e_2重新应用回外层的处理器***h***进行求值，将所求得的值抛给外层的返回延续***r***.