PFPL Part X

Exceptions and Continuations

Linca

2025-09-03

Introduction

> *The technique of structural dynamics is very useful for theoretical purposes, such as proving type safety, but is too high level to be directly usable in an implementation.*

structural dynamics 适合用来进行形式化证明，但是对语言实现而言过于高层. 我们并不知道在当前计算中我们走到了哪一步.

因此我们引入 *control stack* 的概念，用一个显式的数据结构跟踪我们当前在哪里，接下来要做什么.

接下来我们将在 $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ 中这样引入的新的原语. 新的抽象机器我们称之为 $\mathcal{K}\{\texttt{nat} \rightharpoonup\}$

Machine Defintation

A state, s of $\mathcal{K}\{\texttt{nat} \rightharpoonup\}$ consists of a *control stack*, k, and a closed expression, e. States may take one of two formulas

1. An *evaluation* state of the form $k \triangleright e$ corresponds to the evaluation of a closed expression, e, relative to a control stack, k.

含义：我们需要在 control stack k 的上下文中，对 e 进行求值.

2. An *return* state of the form $k \triangleleft e$, where e val, corresponds to the evaluation of a stack, k, relative to a closed value, e.

含义：我们需要将求值好的 e val 返回给 control stack k 的上下文，并继续接下来的求值.

Machine Defintation

Control stack 表示的是当前的计算上下文. 它存储了当前计算的“位置”. 形式化的说，一个 control stack 是一个 *frames* list:

$$\frac{}{\epsilon\ \mathsf{stack}} \tag{27.1a}$$

$$\frac{f\ \mathsf{frame} \quad k\ \mathsf{stack}}{k; f\ \mathsf{stack}} \tag{27.1b}$$

接下来我们将要定义哪些是 frame.

Machine Defintation

$\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ 的 frames 导出地定义为:

$$\frac{}{\texttt{s}(-)\ \mathsf{frame}} \tag{27.2a}$$

$$\frac{}{\texttt{ifz}(-; e_1; x.e_2)\ \mathsf{frame}} \tag{27.2b}$$

$$\frac{}{\texttt{ap}(-, e_2)\ \mathsf{frame}} \tag{27.2c}$$

回忆 Chapter 10.2 的内容，$\mathcal{K}\{\texttt{nat} \rightharpoonup\}$ 的 frams 与 $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ 的 dynamics 是对应的.

Machine Defintation

接下来，让我们从 natural numbers 讲述 rules

$$\frac{}{k \triangleright \mathtt{z} \mapsto k \triangleleft \mathtt{z}} \tag{27.3a}$$

$$\frac{}{k \triangleright \mathtt{s}(e) \mapsto k;\mathtt{s}(-) \triangleright e} \tag{27.3b}$$

$$\frac{}{k;\mathtt{s}(-) \triangleleft e \mapsto k \triangleleft \mathtt{s}(e)} \tag{27.3c}$$

(27.3a) 表示计算 $\mathtt{z}$ 我们将其直接返回. (27.3b) 表示计算 $\mathtt{s}(e)$ 我们创建一个新的 frame 等待内部的 e. (27.3c) 表示当刚刚所说的 frame 返回后我们可以弹出栈顶，继续计算.

Machine Defintation

接下来是 ifz 的规则.

$$\frac{}{k \triangleright \mathtt{ifz}(e; e_1; e_2) \mapsto k;\ \mathtt{ifz}(-; e_1; x.e_2) \triangleleft e} \tag{27.4a}$$

$$\frac{}{k;\ \mathtt{ifz}(-; e_1; x.e_2) \triangleleft \mathtt{z} \mapsto k \triangleright e_1} \tag{27.4b}$$

$$\frac{}{k;\ \mathtt{ifz}(-; e_1; x.e_2) \triangleleft \mathtt{s}(e) \mapsto k \triangleright [e/x]e_2} \tag{27.4c}$$

类似的， (27.4a) 先计算 e 并将 $\mathtt{ifz}(-; e_1; x.e_2)$ 压栈， 然后 (27.4b) 展示了 returns `z` 的情况， (27.4c) 展示了 returns `s` 的情况.

Machine Defintation

最后是 lambda 演算的对应规则.

$$\frac{}{k \triangleright \mathtt{lam}[\tau](x.e) \mapsto k \triangleleft \mathtt{lam}[\tau](x.e)} \tag{27.5a}$$

$$\frac{}{k \triangleright \mathtt{ap}(e_1; e_2) \mapsto k; \mathtt{ap}(-; e_2) \triangleright e_1} \tag{27.5b}$$

$$\frac{}{k; \mathtt{ap}(-; e_2) \triangleleft \mathtt{lam}[\tau](x.e) \mapsto k \triangleright [e_2/x]e} \tag{27.5c}$$

$$\frac{}{k \triangleright \mathtt{fix}[\tau](x.e) \mapsto k \triangleright [\mathtt{fix}[\tau](x.e)/x]e} \tag{27.5d}$$

Machine Defintation

接下来是初始状态和终止状态.

$$\frac{}{\epsilon \triangleright e \text{ initial}} \tag{27.6a}$$

$$\frac{e \text{ val}}{\epsilon \triangleleft e \text{ final}} \tag{27.6b}$$

Machine Defintation

让我们用一个例子，比如计算 $\mathtt{s}(\mathtt{s}(\mathtt{z}))$ 的前驱来理解.

$$
\begin{aligned}
& & \epsilon \triangleright \mathtt{ifz}(\mathtt{s}(\mathtt{s}(\mathtt{z}));\mathtt{z};x.x) \text{ initial} & \\
\mapsto & & \epsilon;\ \mathtt{ifz}(-;\mathtt{z};x.x) \triangleright \mathtt{s}(\mathtt{s}(\mathtt{z})) & \text{using (27.4a)} \\
\mapsto & & \epsilon;\ \mathtt{ifz}(-;\mathtt{z};x.x);\mathtt{s}(-) \triangleright \mathtt{s}(\mathtt{z}) & \text{using (27.3b)} \\
\mapsto & \epsilon; & \mathtt{ifz}(-;\mathtt{z};x.x);\mathtt{s}(-);\mathtt{s}(-) \triangleright \mathtt{z} & \text{using (27.3b)} \\
\mapsto & \epsilon; & \mathtt{ifz}(-;\mathtt{z};x.x);\mathtt{s}(-);\mathtt{s}(-) \triangleleft \mathtt{z} & \text{using (27.3a)} \\
\mapsto & & \epsilon;\ \mathtt{ifz}(-;\mathtt{z};x.x);\mathtt{s}(-) \triangleleft \mathtt{s}(\mathtt{z}) & \text{using (27.3c)} \\
\mapsto & & \epsilon;\ \mathtt{ifz}(-;\mathtt{z};x.x) \triangleleft \mathtt{s}(\mathtt{s}(\mathtt{z})) & \text{using (27.3c)} \\
\mapsto & & \epsilon \triangleright \mathtt{s}(\mathtt{z}) & \text{using (27.3c)} \\
\mapsto & & \epsilon;\mathtt{s}(-) \triangleright [\mathtt{s}(\mathtt{z})/x]x = \mathtt{s}(\mathtt{z}) & \text{using (27.3b)} \\
\mapsto & & \epsilon;\mathtt{s}(-) \triangleleft \mathtt{z} & \text{using (27.3a)} \\
\mapsto & & \epsilon \triangleleft \mathtt{s}(\mathtt{z}) \text{ final} & \text{using (27.3b)}
\end{aligned}
$$

Safety

为了定义和证明 $\mathcal{K}\{\texttt{nat} \rightharpoonup\}$ 的 safety 我们引入新的 typing judgment $k : \tau$ 表示 stack t 期望类型 τ 的值.

$$\frac{}{\epsilon : \tau} \tag{27.7a}$$

$$\frac{k : \tau' \quad f : \tau \Rightarrow \tau'}{k; f : \tau} \tag{27.7b}$$

$f : \tau \Rightarrow \tau'$ 表示栈帧 f 会把 value of type τ 转换成 value of type τ'

Safety

三种 frame 的类型分别如下：

$$\frac{}{\mathtt{s}(-) : \mathtt{nat} \Rightarrow \mathtt{nat}} \tag{27.8a}$$

$$\frac{e_1 : \tau \quad x : \mathtt{nat} \vdash e_2 : \tau}{\mathtt{ifz}(-; e_1; x.e_2) : \mathtt{nat} \Rightarrow \tau} \tag{27.8b}$$

$$\frac{e_2 : \tau}{\mathtt{ap}(-, e_2) : \mathtt{arr}(\tau_2; \tau) \Rightarrow \tau} \tag{27.8c}$$

Safety

The states of $\mathcal{K}\{\texttt{nat} \rightharpoonup\}$ are well-formed if their stack and expression components match:

$$\frac{k : \tau \quad e : \tau}{k \triangleright e \ \mathsf{ok}} \tag{27.9a}$$

$$\frac{k : \tau \quad e : \tau \quad e \ \mathsf{val}}{k \triangleleft e \ \mathsf{ok}} \tag{27.9b}$$

Correctness of the Control Machine

If we evaluate a given expression, e, using $\mathcal{K}\{\texttt{nat} \rightharpoonup\}$, do we get the same result as would be given by $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, and *vice versa*?

对 $\mathcal{K}\{\texttt{nat} \rightharpoonup\}$ 与 $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ 定义 soundness 和 completeness:

Completeness If $e \mapsto^* e'$ where e' val, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e'$.

Soundness If $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e'$, then $e \mapsto^* e'$ with e' val.

Correctness of the Control Machine

从 completeness 讲起. 最终目的是导出 If $e \mapsto^* e'$ where e' val, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e'$. 我们可以先运用数学归纳法:

- e val, then $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft e$

这条证明比较显然，考虑到 val 只有函数（立即返回）和 $\mathsf{s}^n(\mathsf{z})$

- $e \mapsto e'$, then $\forall v$ val, $(\epsilon \triangleright e' \mapsto^* \epsilon \triangleleft v) \to (\epsilon \triangleright e' \mapsto^* \epsilon \triangleleft v)$

这条证明接下来说.

这样对于 $e_1 \mapsto e_2 \mapsto \ldots \mapsto e_n$ val 可以逐步归纳到 $\epsilon \triangleright e_1 \mapsto \epsilon \triangleleft e_n$

Correctness of the Control Machine

A generalization is to prove that if $e \mapsto e'$ and $k \triangleright e' \mapsto^* k \triangleleft v$, then $k \triangleright e \mapsto^* k \triangleleft v$. Consider again the case $e = \mathsf{ap}(e_1; e_2)$, $e' = \mathsf{ap}(e_1'; e_2)$, with $e_1 \mapsto e_1'$. We are given that $k \triangleright \mathsf{ap}(e_1'; e_2) \mapsto^* k \triangleleft v$, and we are to show that $k \triangleright \mathsf{ap}(e_1; e_2) \mapsto^* k \triangleleft v$. It is easy to show that the first step of the former derivation is

$$k \triangleright \mathsf{ap}(e_1'; e_2) \mapsto k; \mathsf{ap}(-; e_2) \triangleright e_1'.$$

We would like to apply induction to the derivation of $e_1 \mapsto e_1'$, but to do so we must have a v_1 such that $e_1' \mapsto^* v_1$, which is not immediately at hand.

This means that we must consider the ultimate value of each sub-expression of an expression in order to complete the proof. This information is provided by the evaluation dynamics described in Chapter 7, which has the property that $e \Downarrow e'$ iff $e \mapsto^* e'$ and e' val.

Correctness of the Control Machine

Lemma 27.2. If $e \Downarrow v$, then for every k stack, $k \triangleright e \mapsto^* k \triangleleft v$

归纳证明它很困难，为了证明它，我们可以考虑每个 $\mathcal{K}\{\texttt{nat} \rightharpoonup\}$ 状态编码了一个表达式，它的 transitions 可以被 $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ transitions 模拟.

引入 $s \leftrightarrows e$ 表示 s 可以被 解释 成 e.

For $s = \epsilon \triangleright e$ initial and $s = \epsilon \triangleleft e$ final, we have $s \leftrightarrows e$

We want to show that $\begin{cases} s \mapsto^* s' \\ s' \text{ final} \\ s \leftrightarrows e \\ s' \leftrightarrows e' \end{cases}$ then e' val, $e \mapsto^* e'$

Correctness of the Control Machine

- If $s \rightarrowtail e$ and s final then e val. 由 (27.6b)
- **Lemma 27.3.** If $s \mapsto s'$, $s \rightarrowtail e$, $s' \rightarrowtail e'$, then $e \mapsto^* e'$

我们需要引入 $k \bowtie e = e'$ 表示既有 $k \triangleright e \rightarrowtail e'$ 又有 $k \triangleleft e \rightarrowtail e'$

$$\overline{\epsilon \bowtie e = e} \tag{27.12a}$$

$$\frac{k \bowtie \mathtt{s}(e) = e'}{k;\mathtt{s}(-) \bowtie e = e'} \tag{27.12b}$$

$$\frac{k \bowtie \mathtt{ifz}(e_1;e_2;x.e_3) = e'}{k;\mathtt{ifz}(-;e_2;x.e_3) \bowtie e_1 = e'} \tag{27.12c}$$

$$\frac{k \bowtie \mathtt{ap}(e_1;e_2) = e}{k;\mathtt{ap}(-;e_2) \bowtie e_1 = e} \tag{27.12d}$$

Correctness of the Control Machine

Lemma 27.5. The judgment $s \mapsto e$ has mode $(\forall, \exists!)$, and the judgment $k \bowtie e = e'$ has mode $(\forall, \forall, \exists!)$

新书的表示是 The judgment $s \mapsto e$ relates every state s to a unique expression e, and the judgment $k \bowtie e = e'$ relates every stack k and expression e to a unique expression e'

Lemma 27.6. If $e \mapsto e', k \bowtie e = d, k \bowtie e' = d'$, then $d \mapsto d'$

我们现在终于能证明 Lemma 27.3.

这样 Completeness 也可以证明：

Correctness of the Control Machine

Proof of Lemma 27.2. The proof is by induction on an evaluation dynamics for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$.

Consider the evaluation rule

$$\frac{e_1 \Downarrow \mathtt{lam}[\tau_2](x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1;e_2) \Downarrow v} \tag{27.10}$$

For an arbitrary control stack, k, we are to show that $k \triangleright \mathtt{ap}(e_1;e_2) \mapsto^* k \triangleleft v$. Applying both of the inductive hypotheses in succession, interleaved with steps of the abstract machine, we obtain

$$\begin{aligned} k \triangleright \mathtt{ap}(e_1;e_2) &\mapsto k;\mathtt{ap}(-;e_2) \triangleright e_1 \\ &\mapsto^* k;\mathtt{ap}(-;e_2) \triangleleft \mathtt{lam}[\tau_2](x.e) \\ &\mapsto k \triangleright [e_2/x]e \\ &\mapsto^* k \triangleleft v. \end{aligned}$$

The other cases of the proof are handled similarly. □

Introduction

Exceptions effect a non-local transfer of control from the point at which the exception is raised to an enclosing handler for that exception.

```
effect raise
  ctl raise( msg : string ) : a
fun safe-divide( x : int, y : int ) : raise int
  if y==0 then raise("div-by-zero") else x / y
fun raise-const() : int
  with handler
    ctl raise(msg) 42
  8 + safe-divide(1,0)
```

Failures

Syntax

Exp e ::=	`fail`	`fail`	failure
	$\texttt{catch}(e_1; e_2)$	$\texttt{catch}\ e_1\ \texttt{ow}\ e_2$	handler

Statics

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \tag{28.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{catch}(e_1; e_2) : \tau} \tag{28.1b}$$

Failures

Failure 可以是任何类型，因为它永远不会返回[1]. 同时这里的 catch 两个类型必须一致，因为 value 可能是任何一个.

Failure 的 dynamics 可以用 *stack unwinding* 来给出. 直观地说，一个 `fail` 将会一路弹出 control stack 的栈顶，直到找到最近的一个 $\texttt{catch}(-; e_2)$，然后计算 e_2 为整个 catch 的 value.

让我们继续用 stack machine 来抽象它. 引入新的 state, k ◀

[1] 所以为什么不干脆让它是 `never` 呢

Failures

在之前的基础上，我们再添加下列规则：

$$\frac{}{k \triangleright \mathtt{fail} \mapsto k \blacktriangleleft} \tag{28.2a}$$

$$\frac{}{k \triangleright \mathtt{catch}(e_1; e_2) \mapsto k; \mathtt{catch}(-; e_2) \triangleright e_1} \tag{28.2b}$$

$$\frac{}{k; \mathtt{catch}(-; e_2) \triangleleft v \mapsto k \triangleleft v} \tag{28.2c}$$

$$\frac{}{k; \mathtt{catch}(-; e_2) \blacktriangleleft \mapsto k \triangleright e_2} \tag{28.2d}$$

$$\frac{}{k; f \blacktriangleleft \mapsto k \blacktriangleleft} \tag{28.2e}$$

Failures

initial state 不变的. 但是， final state 现在还需要包含另一种情况

$$\frac{e\ \mathsf{val}}{\epsilon \lhd e\ \mathsf{final}} \tag{28.3a}$$

$$\frac{}{\epsilon \blacktriangleleft \mathsf{final}} \tag{28.3b}$$

Safety

1. If s ok and $s \mapsto s'$, then s' ok
2. If s ok, then either s final or there exists s' that $s \mapsto s'$

Exceptions

Failure 不承载数据，只是简单的一路 unwind, 我们更多的用 Exception，负载错误对应的数据. 这样我们能分辨是什么错误.

$$\mathsf{Exp}\ e ::= \texttt{raise}[\tau](e) \qquad \texttt{raise}(e) \qquad \text{exception}$$

$$\texttt{handle}(e_1; x.e_2) \quad \texttt{handle}\ e_1\ \texttt{ow}\ x \Rightarrow e_2 \quad \text{handler}$$

Statics

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise}[\tau](e) : \tau} \tag{28.4a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{handle}(e_1; x.e_2) : \tau} \tag{28.4b}$$

Exceptions

类似的，扩展 $k \blacktriangleleft$ 到 $k \blacktriangleleft e$ 其中 $e : \tau_{exn}$ 可以继续得到 dynamics

Dynamics

$$\frac{}{k \triangleright \mathtt{raise}[\tau](e) \mapsto k;\mathtt{raise}[\tau](-) \triangleright e} \tag{28.5a}$$

$$\frac{}{k;\mathtt{raise}[\tau](-) \triangleleft e \mapsto k \blacktriangleleft e} \tag{28.5b}$$

$$\frac{}{k;\mathtt{raise}[\tau](-) \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{28.5c}$$

Exceptions

$$\frac{}{k \triangleright \texttt{handle}(e_1; x\,.\,e_2) \mapsto k;\texttt{handle}(-; x\,.\,e_2) \triangleright e_1} \quad (28.5d)$$

$$\frac{}{k;\texttt{handle}(-; x\,.\,e_2) \triangleleft e \mapsto k \triangleleft e} \quad (28.5e)$$

$$\frac{}{k;\texttt{handle}(-; x\,.\,e_2) \blacktriangleleft e \mapsto k \triangleright [e/x]e_2} \quad (28.5f)$$

$$\frac{(f \neq \texttt{handle}(-; x\,.\,e_2))}{k; f \blacktriangleleft e \mapsto k \blacktriangleleft e} \quad (28.5g)$$

Exception Type

Exceptions 的 statics 用到了 τ_{exn}，它没有限制，但是必须在整个程序中一致.

显然，我们有几个流行的选择:

- `str`，然后就类似于 `raise "404 Not Found"`
- `nat`, 然后用 error numbers 代表错误的唯一 ID[1]
- sum type，最强大的方式. 问题在于，需要分析一个全局一致的类型，整个系统的所有模块都得就异常类型达成一致.

[1]就像 Unix 那样

Encapsulation of Exceptions

有时候我们希望能区分一个 expression 会不会 fail 或者 raise.

An expression is called *fallible*, or *exceptional*, if it can fail or raise an exception during its evaluation, and is *infallible*, or *unexceptional*, otherwise.

To formalize this distinction we distinguish two *modes* of expression, the fallible and the infallible, linked by a *modality* classifying the fallible expressions of a type.

Encapsulation of Exceptions

Type	τ	$::=$	$\texttt{fallible}(\tau)$	$\tau\ \texttt{fallible}$	fallible
Fall	f	$::=$	$\texttt{fail}$	$\texttt{fail}$	failure
			$\texttt{ok}(e)$	$\texttt{ok}(e)$	success
			$\texttt{try}(e; x.f_1; f_2)$	$\texttt{let fall}(x)\ \texttt{be}\ e\ \texttt{in}\ f_1\ \texttt{ow}\ f_2$	handler
Infall	e	$::=$	x	x	variable
			$\texttt{fall}(f)$	$\texttt{fall}(f)$	fallible
			$\texttt{try}(e; x.e_1; e_2)$	$\texttt{let fall}(x)\ \texttt{be}\ e\ \texttt{in}\ e_1\ \texttt{ow}\ e_2$	handler

$\texttt{fallible}(\tau)$ 表示可能失败的表达式 of τ.

infallible expressions 可以包含被 encapsulated fallible expressins.

接下来我们区分 infallible expressions 为 $\Gamma \vdash e : \tau$, 和 fallible 的 $\Gamma \vdash f \sim \tau$

Encapsulation of Exceptions

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{28.6a}$$

$$\frac{\Gamma \vdash f \sim \tau}{\Gamma \vdash \mathtt{fall}(f) : \mathtt{fallible}(\tau)} \tag{28.6b}$$

$$\frac{\Gamma \vdash e : \mathtt{fallible}(\tau) \quad \Gamma, x : \tau \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \mathtt{try}(e; x.e_1; e_2) : \tau'} \tag{28.6c}$$

$$\frac{}{\Gamma \vdash \mathtt{fail} \sim \tau} \tag{28.6d}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{ok}(e) \sim \tau} \tag{28.6e}$$

$$\frac{\Gamma \vdash e : \mathtt{fallible}(\tau) \quad \Gamma, x : \tau \vdash f_1 \sim \tau' \quad \Gamma \vdash f_2 \sim \tau'}{\Gamma \vdash \mathtt{try}(e; x.f_1; f_2) \sim \tau'} \tag{28.6f}$$

Encapsulation of Exceptions

The dynamics of encapsulated failures is readily derived, though some care must be taken with the elimination form for the modality.

$$\frac{}{\mathtt{fall}(f)\ \mathsf{val}} \tag{28.7a}$$

$$\frac{}{k \triangleright \mathtt{try}(e; x.e_1; e_2) \mapsto k; \mathtt{try}(-; x.e_1; e_2) \triangleright e} \tag{28.7b}$$

$$\frac{}{k; \mathtt{try}(-; x.e_1; e_2) \triangleleft \mathtt{fall}(f) \mapsto k; \mathtt{try}(-; x.e_1; e_2); \mathtt{fall}(-) \triangleright f} \tag{28.7c}$$

Encapsulation of Exceptions

$$\frac{}{k \triangleright \mathtt{fail} \mapsto k \blacktriangleleft} \quad (28.7d)$$

$$\frac{}{k \triangleright \mathtt{ok}(e) \mapsto k;\mathtt{ok}(-) \triangleright e} \quad (28.7e)$$

$$\frac{}{k;\mathtt{ok}(-) \triangleleft e \mapsto k \triangleleft \mathtt{ok}(e)} \quad (28.7f)$$

$$\frac{e \text{ val}}{k;\mathtt{try}(-;x.e_1;e_2);\mathtt{fall}(-) \triangleleft \mathtt{ok}(e) \mapsto k \triangleright [e/x]e_1} \quad (28.7g)$$

$$\frac{}{k;\mathtt{try}(-;x.e_1;e_2);\mathtt{fall}(-) \blacktriangleleft \mapsto k \triangleright e_2} \quad (28.7h)$$

Effect

Exceptions 也可以表达为 Effect.

一个有趣的支持 Algebraic Effect 的语言 The Koka Programing Language,可以自行阅读.

另一些 Effect 的效果与接下来的 Continuations 类似.

Introduction

Exception 很有用，但它只能单向地前往 control stack 的底部. 实际上我们有时候需要一些更加灵活的 control flow ， 例如.....

Generators

```
function* miao(): Generator<string, string[], string> {
  const words = ['hello', 'world'];
  const ret: string[] = [];
  for (const word of words) {
    ret.push(yield word);
  }
  return ret;
}
```

Introduction

```
const gen = miao();
console.log(gen.next());
// { value: 'hello', done: false }
console.log(gen.next('hi'));
// { value: 'world', done: false }
console.log(gen.next('everyone'));
// { value: ['hi', 'everyone'], done: true }
```

控制流在调用者和被调用者中来来回回切换.

Introduction

call/cc

```
(define (find-first pred lst)
  (call/cc                              ; 捕获当前 continuation
    (lambda (exit)                      ; exit 是一个“一键回退”的函数
      (for-each
        (lambda (x)
          (when (pred x)
            (exit x)))                  ; 直接跳出整个函数并返回 x
        lst)
      #f)))                             ; 没找到时返回 #f

(find-first even? '(1 3 5 6 7 8))
;; ⇒ 6
```

Informal Overview

We will extend $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with the type $\texttt{cont}(\tau)$ of continuations accepting values of type τ. The introduction form for `cont` is $\texttt{letcc}[\tau](x.e)$, which binds the current continuation (that is, the current control stack) to the variable x, and evaluates the expression e. The corresponding elimination form is $\texttt{throw}[\tau](e_1; e_2)$, which restores the value of e_1 to the control stack that is the value of e_2.

我们以一个函数为例. 给定序列 $q : \texttt{nat} \to \texttt{nat}$, 求它前 n 个元素的积.

Informal Overview

$\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ without short-cutting:

```
fix ms is
  λ q : nat → nat.
    λ n : nat.
      case n {
        z ⇒ s(z)
      | s(n') ⇒ (q z) × (ms (q ∘ succ) n')
      }
```

然而，这个函数可能比较慢. 由于乘法的性质，显然，如果有一个元素 $q(x) = 0$ 则我们可以不用继续计算下去了.

如果 $x \ll n$ 自然地，要是能提前返回，可以加速.

Informal Overview

The version with short-cutting:

```
λ q : nat → nat.
  λ n : nat.
    letcc ret : nat cont in
      let ms be
        fix ms is
          λ q : nat → nat.
            λ n : nat.
              case n {
                z ⇒ s(z)
              | s(n') →
                  case q z {
                    z ⇒ throw z to ret
                  | s(n'') ⇒ (q z) × (ms (q ∘ succ) n')
                  }
              }
      in
        ms q n
```

`letcc` 把当前的 control stack 直接打包给了 `ret`，一旦中间遇到 0，`throw z to ret` 从 `ret` 中读取 control stack 并快速返回.

Semantics of Continuations

We extend the language of $\mathcal{L}\{\rightarrow\}$ expressions with these additional forms:

Type	τ	$::=$	$\texttt{cont}(\tau)$	$\tau\,\texttt{cont}$	continuation
Expr	e	$::=$	$\texttt{letcc}[\tau](x.e)$	$\texttt{letcc}\,x\,\texttt{in}\,e$	mark
			$\texttt{throw}[\tau](e_1;e_2)$	$\texttt{throw}\,e_1\,\texttt{to}\,e_2$	goto
			$\texttt{cont}(k)$	$\texttt{cont}(k)$	continuation

The expression $\texttt{cont}(k)$ is a reified control stack, which arises during evaluation.

Semantics of Continuations

The statics of this extension is defined by the following rules:

$$\frac{\Gamma, x : \texttt{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{letcc}[\tau](x.e) : \tau} \tag{29.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \texttt{cont}(\tau_1)}{\Gamma \vdash \texttt{throw}[\tau'](e_1; e_2) : \tau'} \tag{29.1b}$$

The result type of a `throw` expression is arbitrary because it does not return to the point of the call.

The statics of continuation values is given by the following rule:

$$\frac{k : \tau}{\Gamma \vdash \texttt{cont}(k) : \texttt{cont}(\tau)} \tag{29.2}$$

A continuation value $\texttt{cont}(k)$ has type $\texttt{cont}(\tau)$ exactly if it is a stack accepting values of type τ.

Semantics of Continuations

To define the dynamics we extend $\mathcal{K}\{\mathtt{nat} \rightharpoonup\}$ stacks with two new forms of frame:

$$\frac{}{\mathtt{throw}[\tau](-; e_2)\ \mathsf{frame}} \tag{29.3a}$$

$$\frac{e_1\ \mathsf{val}}{\mathtt{throw}[\tau](e_1; -)\ \mathsf{frame}} \tag{29.3b}$$

Every reified control stack is a value:

$$\frac{k\ \mathsf{stack}}{\mathtt{cont}(k)\ \mathsf{val}} \tag{29.4}$$

Semantics of Continuations

The transition rules for the continuation constructs are as follows:

$$\frac{}{k \triangleright \texttt{letcc}[\tau]\,(x.e) \mapsto k \triangleright [\texttt{cont}(k)/x]e} \tag{29.5a}$$

$$\frac{}{k \triangleright \texttt{throw}[\tau]\,(e_1; e_2) \mapsto k; \texttt{throw}[\tau]\,(-; e_2) \triangleright e_1} \tag{29.5b}$$

$$\frac{e_1 \text{ val}}{k; \texttt{throw}[\tau]\,(-; e_2) \triangleleft e_1 \mapsto k; \texttt{throw}[\tau]\,(e_1; -) \triangleright e_2} \tag{29.5c}$$

$$\frac{}{k; \texttt{throw}[\tau]\,(v; -) \triangleleft \texttt{cont}(k') \mapsto k' \triangleleft v} \tag{29.5d}$$

Evaluation of a `letcc` expression duplicates the control stack; evaluation of a `throw` expression destroys the current control stack.

Semantics of Continuations

The safety of this extension of $\mathcal{L}\{\to\}$ may be established by a simple extension to the safety proof for $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ given in Chapter 27.

We need only add typing rules for the two new forms of frame, which are as follows:

$$\frac{e_2 : \mathtt{cont}(\tau)}{\mathtt{throw}[\tau']\,(-; e_2) : \tau \Rightarrow \tau'} \tag{29.6a}$$

$$\frac{e_1 : \tau \quad e_1\ \mathsf{val}}{\mathtt{throw}[\tau']\,(e_1; -) : \mathtt{cont}(\tau) \Rightarrow \tau'} \tag{29.6b}$$

The rest of the definitions remain as in Chapter 27.

Lemma 29.1 (Canonical Forms). *If $e : \mathtt{cont}(\tau)$ and e val, then $e = \mathtt{cont}(k)$ for some k such that $k : \tau$.*

Theorem 29.2 (Safety). 1. *If s ok and $s \mapsto s'$, then s' ok.*

2. *If s ok, then either s final or there exists s' such that $s \mapsto s'$.*

Coroutines

常规的 subroutine： caller 调用 callee 并等待 callee, callee 只有在完成工作后把结果给 caller. 控制流总是从 caller 到 callee，再返回到 caller.

Coroutine: 两个 routines 彼此调用，共同推进任务. 它们的关系是对称的而非上下层的. 协程 A 运行一段时间后，可以暂停自身并将控制流和当前状态直接移交给协程 B. 协程 B 同样也可以暂停并交还给 A. 控制流在协程之间相互传递.

考虑 coroutine 里每个 routine 的类型是什么?

Coroutines

```
function* my_coroutine() {
  const a = yield 1, b = yield 2; return a + b;
}
```

A routine is a continuation accepting two arguments, a datum to be passed to that routine when it is resumed, and a continuation to be resumed when the routine has finished its task.

$$\tau\ \texttt{coro} \cong (\tau \times \tau\ \texttt{coro})\ \texttt{cont}$$

$$\tau\ \texttt{coro} \triangleq \mu t.(\tau \times t)\ \texttt{cont}$$

Coroutines

A coroutine r passes control to another coroutine r' by evaluating $\texttt{resume}(\langle s, r' \rangle)$.

$\texttt{resume} : \tau \times \tau\ \texttt{coro} \to \tau \times \tau\ \texttt{coro}$

$\lambda(\langle s, r \rangle : \tau \times \tau\ \texttt{coro})\ \texttt{letcc}\ k\ \texttt{in}\ \texttt{throw}\ \langle s, \texttt{fold}(k) \rangle\ \texttt{to}\ \texttt{unfold}\ (r')$

Coroutines

我们可以用 `run` 来启动两个 coroutine:

$\lambda(\langle r_1, r_2\rangle)\lambda(s_0)\ \texttt{letcc}\ x_0\ \texttt{in let}\ r_1'\ \texttt{be}\ r_1(x_0)\ \texttt{in let}\ r_2'\ \texttt{be}\ r_2(x_0)\ \texttt{in}\ldots$

`letcc` 得到二者共同的退出点 x_0，再把这个 continuation 传给二者.

而刚刚函数的 body 又是:

$$\texttt{rep}(r_2')(\texttt{letcc}\ k\ \texttt{in}\ \texttt{rep}(r_1')(\langle s_0, \texttt{fold}(k)\rangle))$$

其中 `rep` 是

$$\lambda(t)\ \texttt{fix}\ l\ \texttt{is}\ \lambda(\langle s, r\rangle)l(\texttt{resume}(\langle t(s), r\rangle))$$

Coroutines

书中举了一个 coroutine 实现 consumer - producer 模型的例子.

This leads to the following implementation of the producer/consumer model. The type τ of the state maintained by the routines is the labeled sum type

$$[\mathtt{OK} \hookrightarrow \tau_i\,\mathtt{list} \times \tau_o\,\mathtt{list}, \mathtt{EMIT} \hookrightarrow \tau_i\,\mathtt{opt} \times (\tau_i\,\mathtt{list} \times \tau_o\,\mathtt{list})].$$

Coroutines

The consumer, C, is defined by the expression

$$\lambda\,(x_0)\;\lambda\,(msg)\;\texttt{case}\,msg\,\{b_1' \mid b_2' \mid b_3'\},$$

where the first branch, b_1', is

$$\texttt{EMIT}\cdot\langle\texttt{null},\langle\_,os\rangle\rangle \Rightarrow \texttt{throw}\,os\,\texttt{to}\,x_0,$$

the second branch, b_2', is

$$\texttt{EMIT}\cdot\langle\texttt{just}(i),\langle is,os\rangle\rangle \Rightarrow \texttt{OK}\cdot\langle is,\texttt{cons}(f(i);os)\rangle,$$

and the third branch, b_3', is

$$\texttt{OK}\cdot\_ \Rightarrow \texttt{error}.$$

The producer, P, is defined by the expression

$$\lambda\,(x_0)\;\lambda\,(msg)\;\texttt{case}\,msg\,\{b_1 \mid b_2 \mid b_3\},$$

where the first branch, b_1, is

$$\texttt{OK}\cdot\langle\texttt{nil},os\rangle \Rightarrow \texttt{EMIT}\cdot\langle\texttt{null},\langle\texttt{nil},os\rangle\rangle$$

and the second branch, b_2, is

$$\texttt{OK}\cdot\langle\texttt{cons}(i;is),os\rangle \Rightarrow \texttt{EMIT}\cdot\langle\texttt{just}(i),\langle is,os\rangle\rangle,$$

and the third branch, b_3, is

$$\texttt{EMIT}\cdot\_ \Rightarrow \texttt{error}.$$

启动计算 $\texttt{run}(\langle P, C\rangle)(s_0)$ 后，两个协程就开始交替执行，直到输入耗尽，返回 os

Threads and Coroutine

对于 $n \geq 2$ participants，直接应用协程没有那么实用. 常见的做法事是多线程调度器.

在这样的 *cooperative multi-threading* 中，每一个 *thread* 都和中心调度器 *scheduler* 是协程.

这些 thread 通过显式的 yield 把控制权让给 scheduler，而 scheduler 再决定接下来执行哪个 thread —— 同样的，也是自身的一个协程.

Thanks

The slides are powered by Typst and touying. University theme