CS 430/530
Formal Semantics

Zhong Shao

Yale University
Department of Computer Science

Control Flow

April 3, 2025

System PCF Dynamics

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{19.2a}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{s}(e)\ \mathsf{val}} \tag{19.2b}$$

$$\frac{}{\mathtt{lam}\{\tau\}(x.e)\ \mathsf{val}} \tag{19.2c}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{s}(e) \longmapsto \mathtt{s}(e')}\right] \tag{19.3a}$$

$$\frac{e \longmapsto e'}{\mathtt{ifz}\{e_0; x.e_1\}(e) \longmapsto \mathtt{ifz}\{e_0; x.e_1\}(e')} \tag{19.3b}$$

$$\frac{}{\mathtt{ifz}\{e_0; x.e_1\}(\mathtt{z}) \longmapsto e_0} \tag{19.3c}$$

$$\frac{\mathtt{s}(e)\ \mathsf{val}}{\mathtt{ifz}\{e_0; x.e_1\}(\mathtt{s}(e)) \longmapsto [e/x]e_1} \tag{19.3d}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1'; e_2)} \tag{19.3e}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1; e_2')}\right] \tag{19.3f}$$

$$\frac{[e_2\ \mathsf{val}]}{\mathtt{ap}(\mathtt{lam}\{\tau\}(x.e); e_2) \longmapsto [e_2/x]e} \tag{19.3g}$$

$$\frac{}{\mathtt{fix}\{\tau\}(x.e) \longmapsto [\mathtt{fix}\{\tau\}(x.e)/x]e} \tag{19.3h}$$

Problems w. PCF Dynamics

Problems w. small-step structural dynamics (transition semantics)

- *It uses rules to decide where to apply the next instruction*
- *Does not say where the instruction lies within an expression*

Abstract Machine K for PCF

A state s of the stack machine **K** for **PCF** consists of a *control stack* k and a closed expression e. States take one of two forms:

1. An *evaluation* state of the form $k \triangleright e$ corresponds to the evaluation of a closed expression e on a control stack k.
2. A *return* state of the form $k \triangleleft e$, where e val, corresponds to the evaluation of a stack k on a closed value e.

Abstract Machine K

The control stack represents the context of evaluation. It records the "current location" of evaluation, the context into which the value of the current expression is returned. Formally, a control stack is a list of *frames*:

$$\frac{}{\epsilon \text{ stack}} \tag{28.1a}$$

$$\frac{f \text{ frame} \quad k \text{ stack}}{k; f \text{ stack}} \tag{28.1b}$$

The frames of the **K** machine are inductively defined by the following rules:

$$\frac{}{\mathtt{s}(-) \text{ frame}} \tag{28.2a}$$

$$\frac{}{\mathtt{ifz}\{e_0; x.e_1\}(-) \text{ frame}} \tag{28.2b}$$

$$\frac{}{\mathtt{ap}(-; e_2) \text{ frame}} \tag{28.2c}$$

Abstract Machine K

The transition judgment between states of the **PCF** machine is inductively defined by a set of inference rules. We begin with the rules for natural numbers, using an eager semantics for the successor.

$$\frac{}{k \triangleright \mathtt{z} \longmapsto k \triangleleft \mathtt{z}} \tag{28.3a}$$

$$\frac{}{k \triangleright \mathtt{s}(e) \longmapsto k;\mathtt{s}(-) \triangleright e} \tag{28.3b}$$

$$\frac{}{k;\mathtt{s}(-) \triangleleft e \longmapsto k \triangleleft \mathtt{s}(e)} \tag{28.3c}$$

Abstract Machine K

Next, we consider the rules for case analysis.

$$\frac{}{k \rhd \mathtt{ifz}\{e_0; x.e_1\}(e) \longmapsto k;\mathtt{ifz}\{e_0; x.e_1\}(-) \rhd e} \tag{28.4a}$$

$$\frac{}{k;\mathtt{ifz}\{e_0; x.e_1\}(-) \lhd \mathtt{z} \longmapsto k \rhd e_0} \tag{28.4b}$$

$$\frac{}{k;\mathtt{ifz}\{e_0; x.e_1\}(-) \lhd \mathtt{s}(e) \longmapsto k \rhd [e/x]e_1} \tag{28.4c}$$

Abstract Machine K

Finally, we give the rules for functions, which are evaluated by-name, and the rule for general recursion.

$$\frac{}{k \triangleright \mathtt{lam}\{\tau\}(x.e) \longmapsto k \triangleleft \mathtt{lam}\{\tau\}(x.e)} \tag{28.5a}$$

$$\frac{}{k \triangleright \mathtt{ap}(e_1; e_2) \longmapsto k;\mathtt{ap}(-; e_2) \triangleright e_1} \tag{28.5b}$$

$$\frac{}{k;\mathtt{ap}(-; e_2) \triangleleft \mathtt{lam}\{\tau\}(x.e) \longmapsto k \triangleright [e_2/x]e} \tag{28.5c}$$

$$\frac{}{k \triangleright \mathtt{fix}\{\tau\}(x.e) \longmapsto k \triangleright [\mathtt{fix}\{\tau\}(x.e)/x]e} \tag{28.5d}$$

Abstract Machine K

The initial and final states of the **K** machine are defined by the following rules:

$$\frac{}{\epsilon \triangleright e \ \mathsf{initial}} \tag{28.6a}$$

$$\frac{e \ \mathsf{val}}{\epsilon \triangleleft e \ \mathsf{final}} \tag{28.6b}$$

Abstract Machine K: Type Safety

To define and prove safety for the **PCF** machine requires that we introduce a new typing judgment, $k \lhd: \tau$, which states that the stack k expects a value of type τ. This judgment is inductively defined by the following rules:

$$\frac{}{\epsilon \lhd: \tau} \tag{28.7a}$$

$$\frac{k \lhd: \tau' \quad f : \tau \rightsquigarrow \tau'}{k; f \lhd: \tau} \tag{28.7b}$$

This definition makes use of an auxiliary judgment, $f : \tau \rightsquigarrow \tau'$, stating that a frame f transforms a value of type τ to a value of type τ'.

$$\frac{}{\mathtt{s}(-) : \mathtt{nat} \rightsquigarrow \mathtt{nat}} \tag{28.8a}$$

$$\frac{e_0 : \tau \quad x : \mathtt{nat} \vdash e_1 : \tau}{\mathtt{ifz}\{e_0; x.e_1\}(-) : \mathtt{nat} \rightsquigarrow \tau} \tag{28.8b}$$

$$\frac{e_2 : \tau_2}{\mathtt{ap}(-; e_2) : \mathtt{parr}(\tau_2; \tau) \rightsquigarrow \tau} \tag{28.8c}$$

Abstract Machine K: Type Safety

The states of the **PCF** machine are well-formed if their stack and expression components match:

$$\frac{k \lhd: \tau \quad e : \tau}{k \rhd e \ \mathsf{ok}} \tag{28.9a}$$

$$\frac{k \lhd: \tau \quad e : \tau \quad e \ \mathsf{val}}{k \lhd e \ \mathsf{ok}} \tag{28.9b}$$

We leave the proof of safety of the **PCF** machine as an exercise.

Theorem 28.1 (Safety)**.** *1. If s ok and $s \longmapsto s'$, then s' ok.*

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

Abstract Machine K: Correctness

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

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

Abstract Machine K: Completeness

To prove completeness a plausible first step is to consider a proof by induction on the definition of multi-step transition, which reduces the theorem to the following two lemmas:

1. If e val, then $\epsilon \triangleright e \longmapsto^* \epsilon \triangleleft e$.
2. If $e \longmapsto e'$, then, for every v val, if $\epsilon \triangleright e' \longmapsto^* \epsilon \triangleleft v$, then $\epsilon \triangleright e \longmapsto^* \epsilon \triangleleft v$.

We therefore consider the value of each sub-expression of an expression. This information is given by the evaluation dynamics described in Chapter 7, which has the property that $e \Downarrow e'$ iff $e \longmapsto^* e'$ and e' val.

Lemma 28.2. *If $e \Downarrow v$, then for every k* stack, *$k \triangleright e \longmapsto^* k \triangleleft v$.*

The desired result follows by the analog of Theorem 7.2 for **PCF**, which states that $e \Downarrow v$ iff $e \longmapsto^* v$.

Abstract Machine K: Completeness

Lemma 28.2. *If $e \Downarrow v$, then for every k* stack, $k \triangleright e \longmapsto^* k \triangleleft v$.

Proof of Lemma 28.2 The proof is by induction on an evaluation dynamics for **PCF**.

Consider the evaluation rule

$$\frac{e_1 \Downarrow \mathtt{lam}\{\tau_2\}(x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1; e_2) \Downarrow v} \tag{28.10}$$

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

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

The other cases of the proof are handled similarly. □

Abstract Machine K: Soundness

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

To do so, we define a judgment, $s \rightarrowtail e$, stating that state s "unravels to" expression e. It will turn out that for initial states, $s = \epsilon \triangleright e$, and final states, $s = \epsilon \triangleleft e$, we have $s \rightarrowtail e$. Then we show that if $s \longmapsto^* s'$, where s' final, $s \rightarrowtail e$, and $s' \rightarrowtail e'$, then e' val and $e \longmapsto^* e'$. For this, it is enough to show the following two facts:

1. If $s \rightarrowtail e$ and s final, then e val.
2. If $s \longmapsto s'$, $s \rightarrowtail e$, $s' \rightarrowtail e'$, and $e' \longmapsto^* v$, where v val, then $e \longmapsto^* v$.

Lemma 28.3. *If $s \longmapsto s'$, $s \rightarrowtail e$, and $s' \rightarrowtail e'$, then $e \longmapsto^* e'$.*

Abstract Machine K: Soundness

The judgment $s \mapsto e'$, where s is either $k \triangleright e$ or $k \triangleleft e$, is defined in terms of the auxiliary judgment $k \bowtie e = e'$ by the following rules:

$$\frac{k \bowtie e = e'}{k \triangleright e \mapsto e'} \tag{28.11a}$$

$$\frac{k \bowtie e = e'}{k \triangleleft e \mapsto e'} \tag{28.11b}$$

In words, to unravel a state, we wrap the stack around the expression to form a complete program. The unraveling relation is inductively defined by the following rules:

$$\frac{}{\epsilon \bowtie e = e} \tag{28.12a}$$

$$\frac{k \bowtie \mathtt{s}(e) = e'}{k;\mathtt{s}(-) \bowtie e = e'} \tag{28.12b}$$

$$\frac{k \bowtie \mathtt{ifz}\{e_0; x.e_1\}(e) = e'}{k;\mathtt{ifz}\{e_0; x.e_1\}(-) \bowtie e = e'} \tag{28.12c}$$

$$\frac{k \bowtie \mathtt{ap}(e_1; e_2) = e}{k;\mathtt{ap}(-; e_2) \bowtie e_1 = e} \tag{28.12d}$$

Abstract Machine K: Soundness

Lemma 28.5. *The judgment $s \mathrel{\rightarrowtail} e$ 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 28.6. *If $e \longmapsto e'$, $k \bowtie e = d$, $k \bowtie e' = d'$, then $d \longmapsto d'$.*

Proof The proof is by rule induction on the transition $e \longmapsto e'$. The inductive cases, where the transition rule has a premise, follow easily by induction. The base cases, where the transition is an axiom, are proved by an inductive analysis of the stack k.

For an example of an inductive case, suppose that $e = \mathtt{ap}(e_1; e_2)$, $e' = \mathtt{ap}(e_1'; e_2)$, and $e_1 \longmapsto e_1'$. We have $k \bowtie e = d$ and $k \bowtie e' = d'$. It follows from rules (28.12) that $k;\mathtt{ap}(-; e_2) \bowtie e_1 = d$ and $k;\mathtt{ap}(-; e_2) \bowtie e_1' = d'$. So by induction $d \longmapsto d'$, as desired.

For an example of a base case, suppose that $e = \mathtt{ap}(\mathtt{lam}\{\tau_2\}(x.e); e_2)$ and $e' = [e_2/x]e$ with $e \longmapsto e'$ directly. Assume that $k \bowtie e = d$ and $k \bowtie e' = d'$; we are to show that $d \longmapsto d'$. We proceed by an inner induction on the structure of k. If $k = \epsilon$, the result follows immediately. Consider, say, the stack $k = k';\mathtt{ap}(-; c_2)$. It follows from rules (28.12) that $k' \bowtie \mathtt{ap}(e; c_2) = d$ and $k' \bowtie \mathtt{ap}(e'; c_2) = d'$. But by the structural dynamics $\mathtt{ap}(e; c_2) \longmapsto \mathtt{ap}(e'; c_2)$, so by the inner inductive hypothesis we have $d \longmapsto d'$, as desired. □

Abstract Machine K: Soundness

Lemma 28.3. *If* $s \longmapsto s'$, $s \mapsto e$, *and* $s' \mapsto e'$, *then* $e \longmapsto^* e'$.

Proof of Lemma 28.3 The proof is by case analysis on the transitions of the **K** machine. In each case, after unraveling, the transition will correspond to zero or one transitions of the **PCF** structural dynamics.

Suppose that $s = k \triangleright \mathtt{s}(e)$ and $s' = k;\mathtt{s}(-) \triangleright e$. Note that $k \bowtie \mathtt{s}(e) = e'$ iff $k;\mathtt{s}(-) \bowtie e = e'$, from which the result follows immediately.

Suppose that $s = k;\mathtt{ap}(\mathtt{lam}\{\tau\}(x.e_1); -) \triangleleft e_2$ and $s' = k \triangleright [e_2/x]e_1$. Let e' be such that $k;\mathtt{ap}(\mathtt{lam}\{\tau\}(x.e_1); -) \bowtie e_2 = e'$ and let e'' be such that $k \bowtie [e_2/x]e_1 = e''$. Observe that $k \bowtie \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e_1); e_2) = e'$. The result follows from Lemma 28.6.

□

FPCF: PCF with Failures

The syntax of **FPCF** is defined by the following extension of the grammar of **PCF**:

Exp	e	::=	`fail`	`fail`	signal a failure
			`catch`$(e_1; e_2)$	`catch` e_1 `ow` e_2	catch a failure

The expression `fail` aborts the current evaluation, and the expression `catch`$(e_1; e_2)$ catches any failure in e_1 by evaluating e_2 instead. Either e_1 or e_2 may themselves abort, or they may diverge or return a value as usual in **PCF**.

The statics of **FPCF** is given by these rules:

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \tag{29.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{catch}(e_1; e_2) : \tau} \tag{29.1b}$$

A failure can have any type, because it never returns. The two expressions in a `catch` expression must have the same type, because either might determine the value of that expression.

FPCF Dynamics: Stack Unwinding

The dynamics of **FPCF** is given using a technique called *stack unwinding*. Evaluation of a `catch` pushes a frame of the form $\texttt{catch}(-; e)$ onto the control stack that awaits the arrival of a failure. Evaluation of a `fail` expression pops frames from the control stack until it reaches a frame of the form $\texttt{catch}(-; e)$, at which point the frame is removed from the stack and the expression e is evaluated. Failure propagation is expressed by a state of the form $k \blacktriangleleft$, which extends the two forms of state considered in Chapter 28 to express failure propagation.

FPCF Dynamics: Stack Unwinding

The **FPCF** machine extends the **PCF** machine with the following additional rules:

$$\frac{}{k \triangleright \texttt{fail} \longmapsto k \blacktriangleleft} \tag{29.2a}$$

$$\frac{}{k \triangleright \texttt{catch}(e_1; e_2) \longmapsto k;\texttt{catch}(-; e_2) \triangleright e_1} \tag{29.2b}$$

$$\frac{}{k;\texttt{catch}(-; e_2) \triangleleft v \longmapsto k \triangleleft v} \tag{29.2c}$$

$$\frac{}{k;\texttt{catch}(-; e_2) \blacktriangleleft \longmapsto k \triangleright e_2} \tag{29.2d}$$

$$\frac{(f \neq \texttt{catch}(-; e))}{k; f \blacktriangleleft \longmapsto k \blacktriangleleft} \tag{29.2e}$$

FPCF Type Safety

The initial and final states of the **FPCF** machine are defined by the following rules:

$$\frac{}{\epsilon \ \mathsf{initial}} \tag{29.3a}$$

$$\frac{e \ \mathsf{val}}{\epsilon \triangleleft e \ \mathsf{final}} \tag{29.3b}$$

$$\frac{}{\epsilon \blacktriangleleft \ \mathsf{final}} \tag{29.3c}$$

The definition of stack typing given in Chapter 28 can be extended to account for the new forms of frame so that safety can be proved in the same way as before. The only difference is that the statement of progress must be weakened to take account of failure: a well-typed expression is either a value, or may take a step, or may signal failure.

Theorem 29.1 (Safety for **FPCF**). *1. If s ok and $s \longmapsto s'$, then s' ok.*

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

XPCF: PCF with Exceptions

The language **XPCF** enriches **FPCF** with *exceptions*, failures to which a value is attached. The syntax of **XPCF** extends that of **PCF** with the following forms of expression:

Exp	e	::=	$\mathtt{raise}(e)$	$\mathtt{raise}(e)$	raise an exception
			$\mathtt{try}(e_1; x.e_2)$	$\mathtt{try}\, e_1\, \mathtt{ow}\, x \hookrightarrow e_2$	handle an exception

The argument to `raise` is evaluated to determine the value passed to the handler. The expression $\mathtt{try}(e_1; x.e_2)$ binds a variable x in the handler e_2. The associated value of the exception is bound to that variable within e_2, should an exception be raised when e_1 is evaluated.

The statics of exceptions extends the statics of failures to account for the type of the value carried with the exception:

$$\frac{\Gamma \vdash e : \tau_{\mathsf{exn}}}{\Gamma \vdash \mathtt{raise}(e) : \tau} \tag{29.4a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{\mathsf{exn}} \vdash e_2 : \tau}{\Gamma \vdash \mathtt{try}(e_1; x.e_2) : \tau} \tag{29.4b}$$

XPCF: PCF with Exceptions

The stack frames of the **PCF** machine are extended to include $\texttt{raise}(-)$ and $\texttt{try}(-; x.e_2)$. These are used in the following rules:

$$\frac{}{k \triangleright \texttt{raise}(e) \longmapsto k;\texttt{raise}(-) \triangleright e} \tag{29.5a}$$

$$\frac{}{k;\texttt{raise}(-) \triangleleft e \longmapsto k \blacktriangleleft e} \tag{29.5b}$$

$$\frac{}{k \triangleright \texttt{try}(e_1; x.e_2) \longmapsto k;\texttt{try}(-; x.e_2) \triangleright e_1} \tag{29.5c}$$

$$\frac{}{k;\texttt{try}(-; x.e_2) \triangleleft e \longmapsto k \triangleleft e} \tag{29.5d}$$

$$\frac{}{k;\texttt{try}(-; x.e_2) \blacktriangleleft e \longmapsto k \triangleright [e/x]e_2} \tag{29.5e}$$

$$\frac{(f \neq \texttt{try}(-; x.e_2))}{k; f \blacktriangleleft e \longmapsto k \blacktriangleleft e} \tag{29.5f}$$

XPCF: PCF with Exceptions

The initial and final states of the **XPCF** machine are defined by the following rules:

$$\frac{}{\epsilon \triangleright e \ \mathsf{initial}} \tag{29.6a}$$

$$\frac{e \ \mathsf{val}}{\epsilon \triangleleft e \ \mathsf{final}} \tag{29.6b}$$

$$\frac{}{\epsilon \blacktriangleleft e \ \mathsf{final}} \tag{29.6c}$$

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

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

Exception Values

This fact suggests that τ_{exn} should be a finite sum. The classes of the sum identify the sources of exceptions, and the classified value carries information about the particular instance. For example, τ_{exn} might be a sum type of the form

$$[\mathtt{div} \hookrightarrow \mathtt{unit}, \mathtt{fnf} \hookrightarrow \mathtt{string}, \ldots].$$

Here the class `div` might represent an arithmetic fault, with no associated data, and the class `fnf` might represent a "file not found" error, with associated data being the name of the file that was not found.

Using a sum means that an exception handler can dispatch on the class of the exception value to identify its source and cause. For example, we might write

$$\begin{array}{l}\mathtt{handle}\ e_1\ \mathtt{ow}\ \mathtt{x} \hookrightarrow \\ \quad \mathtt{match}\ \mathtt{x}\ \{ \\ \quad\quad \mathtt{div}\ \langle\rangle \hookrightarrow e_{\mathsf{div}} \\ \quad \mid \mathtt{fnf}\ s \hookrightarrow e_{\mathsf{fnf}}\ \}\end{array}$$

to handle the exceptions specified by the above sum type. Because the exception and its associated data are coupled in a sum type, there is no possibility of misinterpreting the data associated to one exception as being that of another.

KPCF: PCF with Continuations

The semantics of many control constructs (such as exceptions and coroutines) can be expressed in terms of *reified* control stacks, a representation of a control stack as a value that can be reactivated at any time, *even if* control has long since returned past the point of reification. Reified control stacks of this kind are called *continuations*; they are values that can be passed and returned at will in a computation. Continuations never "expire", and it is always sensible to reinstate a continuation without compromising safety. Thus continuations support unlimited "time travel" — we can go back to a previous step of the computation, then return to some point in its future.

We will consider the extension **KPCF** of **PCF** with the type $\texttt{cont}(\tau)$ of continuations accepting values of type τ. The introduction form for $\texttt{cont}(\tau)$ 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 given by e_1 to the control stack given by e_2.

Motivating Example in PCF

To illustrate the use of these primitives, consider the problem of multiplying the first n elements of an infinite sequence q of natural numbers, where q is represented by a function of type `nat` $\rightharpoonup$ `nat`. If zero occurs among the first n elements, we would like to effect an "early return" with the value zero, without further multiplication. This problem can be solved using exceptions, but we will solve it with continuations to show how they are used.

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

Motivating Example in KPCF w. Short-Cutting

```
λ q : nat ⇀ nat.
  λ n : nat.
    letcc ret : nat cont in
      let m  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
        m  q n
```

Motivating Example in KPCF

To take another example, given that k has type τ `cont` and f has type $\tau' \rightharpoonup \tau$, return a continuation k' of type τ' `cont` such that throwing a value v' of type τ' to k' throws the value of $f(v')$ to k. Thus, we seek to define a function `compose` of type

$$(\tau' \rightharpoonup \tau) \rightharpoonup \tau \texttt{ cont} \rightharpoonup \tau' \texttt{ cont}.$$

The continuation we seek is the one in effect at the point of the ellipsis in the expression `throw` f`(...) to` k. It is the continuation that, when given a value v', applies f to it, and throws the result to k. We can seize this continuation using `letcc` by writing

```
throw f(letcc x:τ' cont in ...) to k
```

The desired continuation is bound to x, but how can we return it as the result of `compose`? We use the same idea as for short-circuit multiplication, writing

```
letcc ret:τ' cont cont in
  throw (f (letcc r in throw r to ret)) to k
```

as the body of `compose`. Note that the type of `ret` is τ'`cont cont`, that of a continuation that expects to be thrown a continuation!

KPCF Syntax & Statics

The syntax of **KPCF** is as follows:

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.

The statics of **KPCF** is defined by the following rules:

$$\frac{\Gamma, x : \texttt{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{letcc}\{\tau\}(x.e) : \tau} \tag{30.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \texttt{cont}(\tau_1)}{\Gamma \vdash \texttt{throw}\{\tau\}(e_1; e_2) : \tau} \tag{30.1b}$$

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:

$$\frac{k : \tau}{\Gamma \vdash \texttt{cont}(k) : \texttt{cont}(\tau)} \tag{30.2}$$

KPCF Dynamics

To define the dynamics of **KPCF**, we extend the **PCF** machine with two forms of stack frame:

$$\frac{}{\texttt{throw}\{\tau\}(-; e_2)\ \mathsf{frame}} \tag{30.3a}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{throw}\{\tau\}(e_1; -)\ \mathsf{frame}} \tag{30.3b}$$

Every reified control stack is a value:

$$\frac{k\ \mathsf{stack}}{\texttt{cont}(k)\ \mathsf{val}} \tag{30.4}$$

KPCF Dynamics

The transition rules of the **PCF** machine governing continuations are as follows:

$$\frac{}{k \triangleright \mathtt{cont}(k) \longmapsto k \triangleleft \mathtt{cont}(k)} \tag{30.5a}$$

$$\frac{}{k \triangleright \mathtt{letcc}\{\tau\}(x.e) \longmapsto k \triangleright [\mathtt{cont}(k)/x]e} \tag{30.5b}$$

$$\frac{}{k \triangleright \mathtt{throw}\{\tau\}(e_1; e_2) \longmapsto k;\mathtt{throw}\{\tau\}(-; e_2) \triangleright e_1} \tag{30.5c}$$

$$\frac{e_1 \text{ val}}{k;\mathtt{throw}\{\tau\}(-; e_2) \triangleleft e_1 \longmapsto k;\mathtt{throw}\{\tau\}(e_1; -) \triangleright e_2} \tag{30.5d}$$

$$\frac{e \text{ val}}{k;\mathtt{throw}\{\tau\}(e; -) \triangleleft \mathtt{cont}(k') \longmapsto k' \triangleleft e} \tag{30.5e}$$

KPCF Type Safety

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

$$\frac{e_2 : \mathtt{cont}(\tau)}{\mathtt{throw}\{\tau'\}(-; e_2) : \tau \rightsquigarrow \tau'} \tag{30.6a}$$

$$\frac{e_1 : \tau \quad e_1 \ \mathsf{val}}{\mathtt{throw}\{\tau'\}(e_1; -) : \mathtt{cont}(\tau) \rightsquigarrow \tau'} \tag{30.6b}$$

The rest of the definitions remain as in Chapter 28.

Lemma 30.1 (Canonical Forms)**.** *If* $e : \mathtt{cont}(\tau)$ *and* e val*, then* $e = \mathtt{cont}(k)$ *for some* k *such that* $k : \tau$*.*

Theorem 30.2 (Safety)**.** *1. If* s ok *and* $s \longmapsto s'$*, then* s' ok*.*

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