PFPL Supplement: Exceptions: Control and Data*

Robert Harper

Fall, 2019

1 Introduction

This note extends **PCFv** given in Harper (2019) to account for two aspects of exceptions:

1. *Control* aspect: normal and exceptional returns.

2. *Data* aspect: exception values.

The first aspect extends **PCFv** to account for both "normal" and "exceptional" returns. The elimination form for the computation type is enriched to account not only for the normal return, but also the exceptional return.[1] The exception return carries with it a value of an unspecified type `exn`. The second aspect defines the type `exn` to be the type `clsfd` of dynamically classified values. In this setting dynamic classifiers label exception values with the form of exception and associate to a value of the corresponding type. Dynamic classification for exceptions is a good choice because it naturally supports modular composition. Depending on how an exception is *spelled* is anti-modular, because two components might have an exception named `error`, spelled thus. But in fact those two exceptions, which happen to be called the same thing, should be distinct. To achieve that requires binding and scope so that, by α-conversion, the spelling of a name never matters.

2 Exception Mechanism

The language **PCFv** is extended with a construct for raising a value of some unspecified type `exn`, and the elimination form for the computation modality is extended to account for both the normal return and the exceptional return of a computation. The statics of this extension is given in Figure 1, and the dynamics in Figure 2. Both are formulated with states of the form $\nu\,\Sigma\,\{\,e\,\}$ to account for exception values in the next section. Type safety (progress and preservation) is readily established using methods considered in the text.

3 Exception Values

The type `exn` of exception values is defined to be the type `clsfd` of dynamically classified values. The idea is that the class of an exception value determines the type of data associated with that

[1] This elegant formulation of exception handling was introduced by Benton and Kennedy (2001).

$$\frac{\Gamma \vdash_\Sigma e : \texttt{exn}}{\Gamma \vdash_\Sigma \texttt{raise}(\, e \,) \stackrel{\cdot}{\sim} \tau}\ \textsc{Raise} \qquad \frac{\Gamma \vdash_\Sigma e_1 : \texttt{comp}(\, \tau \,) \quad \Gamma, x : \tau \vdash_\Sigma e_2 \stackrel{\cdot}{\sim} \tau' \quad \Gamma, x : \texttt{exn} \vdash_\Sigma e_3 \stackrel{\cdot}{\sim} \tau'}{\Gamma \vdash_\Sigma \texttt{bnd}[\, \tau \,](\, e_1 \,;\, x \,.\, e_2 \,;\, x \,.\, e_3 \,) \stackrel{\cdot}{\sim} \tau'}\ \textsc{Try}$$

Figure 1: **PCFv** with Exceptions: Statics

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\, \Sigma \,\{\, \texttt{raise}(\, e \,) \,\}\ \mathsf{final}}\ \textsc{Raise-Final} \qquad \frac{\nu\, \Sigma \,\{\, e \,\} \longmapsto \nu\, \Sigma' \,\{\, e' \,\}}{\nu\, \Sigma \,\{\, \texttt{raise}(\, e \,) \,\} \longmapsto \nu\, \Sigma' \,\{\, \texttt{raise}(\, e' \,) \,\}}\ \textsc{Raise-Arg}$$

$$\frac{\nu\, \Sigma \,\{\, e_1 \,\} \longmapsto \nu\, \Sigma' \,\{\, e_1' \,\}}{\nu\, \Sigma \,\{\, \texttt{bnd}[\, \tau \,](\, e_1 \,;\, x \,.\, e_2 \,;\, x \,.\, e_3 \,) \,\} \longmapsto \nu\, \Sigma' \,\{\, \texttt{bnd}[\, \tau \,](\, e_1' \,;\, x \,.\, e_2 \,;\, x \,.\, e_3 \,) \,\}}\ \textsc{Let}$$

$$\frac{\nu\, \Sigma \,\{\, e \,\} \longmapsto \nu\, \Sigma' \,\{\, e' \,\}}{\nu\, \Sigma \,\{\, \texttt{bnd}[\, \tau \,](\, \texttt{comp}(\, e \,) \,;\, x \,.\, e_2 \,;\, x \,.\, e_3 \,) \,\} \longmapsto \nu\, \Sigma' \,\{\, \texttt{bnd}[\, \tau \,](\, \texttt{comp}(\, e' \,) \,;\, x \,.\, e_2 \,;\, x \,.\, e_3 \,) \,\}}\ \textsc{Let-Comp}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\, \Sigma \,\{\, \texttt{bnd}[\, \tau \,](\, \texttt{comp}(\, \texttt{ret}(\, e \,) \,) \,;\, x \,.\, e_2 \,;\, x \,.\, e_3 \,) \,\} \longmapsto \nu\, \Sigma \,\{\, \{e/x\}e_2 \,\}}\ \textsc{Let-Ret}$$

$$\frac{e\ \mathsf{val}_\Sigma}{\nu\, \Sigma \,\{\, \texttt{bnd}[\, \tau \,](\, \texttt{comp}(\, \texttt{raise}(\, e \,) \,) \,;\, x \,.\, e_2 \,;\, x \,.\, e_3 \,) \,\} \longmapsto \nu\, \Sigma \,\{\, \{e/x\}e_3 \,\}}\ \textsc{Let–Raise}$$

Figure 2: **PCFv** with Exceptions: Dynamics

$$\frac{\Gamma, x : \texttt{exc}(\tau) \vdash_\Sigma e' \mathrel{\dot\sim} \tau'}{\Gamma \vdash_\Sigma \texttt{dclexc}[\tau](x . e') \mathrel{\dot\sim} \tau'} \text{ EXC-DECL}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \texttt{exc}(\tau_2) \qquad \Gamma \vdash_\Sigma e_2 : \tau_2}{\Gamma \vdash_\Sigma \texttt{raiseexc}(e_1 ; e_2) \mathrel{\dot\sim} \tau} \text{ EXC-RAISE}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \texttt{comp}(\tau) \qquad \Gamma \vdash_\Sigma e_2 : \texttt{exc}(\tau_2) \qquad \Gamma, x : \tau_2 \vdash_\Sigma e_3 \mathrel{\dot\sim} \tau}{\Gamma \vdash_\Sigma \texttt{hdlexc}(e_1 ; e_2 ; x . e_3) \mathrel{\dot\sim} \tau} \text{ EXC-HANDLE}$$

Figure 3: Derived Exceptions: Statics

exception class. Exception classes are introduced using an exception declaration; an exception is raised by specifying its class; and an exception is handled by dispatching on the class.

Consider following syntactic extension to **PCFv**:

$$e ::= \texttt{dclexc}[\tau](x . e) \mid \texttt{raiseexc}(e_1 ; e_2) \mid \texttt{hdlexc}(e_1 ; e_2 ; x . e_3)$$

The exception declaration $\texttt{dclexc}[\tau](x . e)$ declares a new exception class, x, with associated type τ for use within e. Notice that x is a *variable* that will be replaced by a class reference within the scope e; its name is immaterial because it is bound within e. The underlying class name is implicitly bound when the class is allocated, so it, too, is different from any other exception class allocated anywhere in a program. *To ensure compositional behavior of exceptions it is essential that exception class declaration incur a run-time storage effect, the allocation of the class name.*

The raise expression $\texttt{raiseexc}(e_1 ; e_2)$ raises an exception value constructed by applying the class reference given by e_1 to the associated value given by e_2 to obtain a value of type `clsfd`, the `exn` type. The handle expression $\texttt{hdlexc}(e_1 ; e_2 ; y . e_3)$ evaluates the computation given by e_1. If it returns normally, that is the value returned by the expression; if it raises an exception of class given by e_2, then it binds the associated value to y within e_3; if it raises any other exception value, then the exception value is re-raised to the surrounding context.

The statics of these constructs is given in Figures 3. It is possible to give a dynamics as well, but instead the dynamics is specified by defining these constructs in terms of dynamic classification, class references, and the control mechanism given in the previous section (see Figure 4.) The exception declaration introduces a new class, a, with the associated type given, and propagates a reference to it into the body of the declaration, a computation of some type. Raising an exception constructed from a class reference and a value of its associated type results creates a classified value to be raised in the sense of Section 2. Handling an exception uses the composite sequencing construct of lax logic to evaluate the given computation, returning its value in the case of a normal return. In the case of an exceptional return the exception value is tested against the given class reference, either propagating the associated value to the handler, or re-raising the exception, according to whether the test succeeds or fails.

4 Summary

There are two aspects of an exception mechanism, the control aspect and the data aspect. The control aspect is neatly accouned for in **PCFv** by considering that an encapsulated computation can complete in one of two ways, by returning a value, or by raising an exception. The elimination form

$$
\begin{aligned}
\texttt{exn} &\triangleq \texttt{clsfd} \\
\texttt{exc}(\tau) &\triangleq \texttt{cls}(\tau) \\
\texttt{dclexc}[\tau](x\,.\,e') &\triangleq \texttt{newcls}[\tau](a\,.\,\{\texttt{cls}\langle a\rangle/x\}e') \\
\texttt{raiseexc}(e_1\,;e_2) &\triangleq \texttt{raise}(\texttt{inref}(e_1\,;e_2)) \\
\texttt{hdlexc}(e_1\,;e_2\,;x\,.\,e_3) &\triangleq \texttt{bnd}[\tau](e_1\,;y\,.\,\texttt{ret}(y)\,;y\,.\,\texttt{isinref}(y\,;e_2\,;x\,.\,e_3\,;\texttt{raise}(y)))
\end{aligned}
$$

Figure 4: Derived Exceptions: Dynamics

for the computation type accounts for both of these outcomes in a natural way by providing both a *normal* continuation and an *exceptional* continuation.

In contrast the control aspect is problematic in by-name **PCF**, because using exceptions relies on a clear understanding control flow, which is absent in by-name languages. For example, $\texttt{s}(\texttt{raise}(e))$ is a value in by-name **PCF**. The encapsulated exception is raised only when the predecessor is evaluated, and not otherwise. The successor is a data structure that, like a land mine, lays in wait to be triggered by an unsuspecting party who is in no way responsible for its origin. Worse, application of a function to an argument that may raise an exception strews the argument throughout the body of the function wherein it may never be evaluated, or evaluated more than once. It is a clear example of a iatrogenic disorder; it is a morass from which there is no escape.

The data aspect accounts for the type of value to be raised when an exception is to be signalled. The most useful choice for the exception value type is the type `clsfd` of *classified* values whose elements are values of an arbitrary type τ labeled with a dynamically generated class, the class of exception being raised. By the miracle of α-equivalence a dynamically generated class is an unguessable secret known only to those program components that have the constructor and destructor associated with it. In particular, when exception values are dynamically classified, the class of the exception is a shared secret between raisers and handlers that have access to that class. Thus, exception raising is not, *pace* the methodology literature that denigrates it, an uncontrolled transfer of control, but the exact opposite. By controlling who has access to the destructor for the class, the raiser can limit the possible handlers for it; correspondingly, by controlling the components that have access to the constructor for the class, the handler can control which components may raise an exception handled by it.

Achieving secrecy requires dynamic generation of exception classes, a computational effect. Thus, pure languages cannot provide the guarantees provided by the secrecy of exception classes. The effect of exception class generation is essential for modularity; without it, two components may accidentally use the same exception class, resulting in chaos. The generation effect is essential, and cannot be sacrificed.

References

Nick Benton and Andrew Kennedy. Exceptional syntax. *Journal of Functional Programming*, 11 (4):395–410, 2001. doi: 10.1017/S0956796801004099.

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, England, Second edition, 2016.

Robert Harper. PCF-By-Value. Supplement to Harper (2016), Fall 2019. URL `https://www.cs.cmu.edu/~rwh/pfpl/supplements/pcfv.pdf`.