PFPL Supplement: Exceptions By Value

Robert Harper

1 Introduction

This note extends the by-value formulation of \textbf{PCF} (Harper, 2018) to account for two aspects of exceptions:

1. \textit{Control} aspect: normal and exceptional returns.
2. \textit{Data} aspect: exception values.

The first aspect extends the modal formulation of \textbf{PCF}-by-value 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. The exception return carries with it a value of an unspecified type \texttt{exn}. The second aspect defines the type \texttt{exn} to be the type \texttt{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: it never matters what an exception is \textit{called}, it only matters what an exception \textit{is}, which is determined at the point at which it is declared. By the miracle of \(\alpha\)-renaming exceptions from different components of a program can never clash when combined.

2 Exception Mechanism

The modal formulation of \textbf{PCF}-by-value is extended with a construct for raising a value of some unspecified type \texttt{exn}, and the elimination form for the 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 \texttt{exn} of exception values is defined to be the type \texttt{clsfd} of dynamically classified values. The idea is that the class of an exception value determines the type of data associated with that 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 modal \textbf{PCF}:

\[
e ::= \texttt{dclexc}\{\tau\}(x.e) | \texttt{raiseexc}(e_1; e_2) | \texttt{handlexc}(e_1; e_2; y.e_3)
\]
RAISE

\[ \Gamma \vdash \Sigma \ \text{exn} \]

\[ \Gamma \vdash \Sigma \ \text{raise}(e) \sim \tau \]

TRY

\[ \Gamma \vdash \Sigma \ e_1 : \text{comp}(\tau) \quad \Gamma, x : \tau \vdash \Sigma \ e_2 \sim \tau' \quad \Gamma, x : \text{exn} \vdash \Sigma \ e_3 \sim \tau' \]

\[ \Gamma \vdash \Sigma \ \text{let}(\tau)\{e_1; x.e_2; x.e_3\} \sim \tau' \]

Figure 1: Modal PCF with Exceptions: Statics

<table>
<thead>
<tr>
<th>RAISE-FINAL</th>
<th>RAISE-ARG</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ e \ \text{val}_\Sigma ]</td>
<td>[ \nu \Sigma { e } \mapsto \nu \Sigma' { e' } ]</td>
</tr>
<tr>
<td>[ \nu \Sigma { \text{raise}(e) } \ \text{final} ]</td>
<td>[ \nu \Sigma { \text{raise}(e) } \mapsto \nu \Sigma' { \text{raise}(e') } ]</td>
</tr>
</tbody>
</table>

LET

\[ \nu \Sigma \{ e_1 \} \mapsto \nu \Sigma' \{ e'_1 \} \]

\[ \nu \Sigma \{ \text{let}(\tau)(e_1; x.e_2; x.e_3) \} \mapsto \nu \Sigma' \{ \text{let}(\tau)(e'_1; x.e_2; x.e_3) \} \]

LET-COMP

\[ \nu \Sigma \{ e \} \mapsto \nu \Sigma' \{ e' \} \]

\[ \nu \Sigma \{ \text{let}(\tau)(\text{comp}(e); x.e_2; x.e_3) \} \mapsto \nu \Sigma' \{ \text{let}(\tau)(\text{comp}(e'); x.e_2; x.e_3) \} \]

LET-RET

\[ e \ \text{val}_\Sigma \]

\[ \nu \Sigma \{ \text{let}(\tau)(\text{comp}(\text{ret}(e)); x.e_2; x.e_3) \} \mapsto \nu \Sigma \{ [e/x]e_2 \} \]

LET-RAISE

\[ e \ \text{val}_\Sigma \]

\[ \nu \Sigma \{ \text{let}(\tau)(\text{comp}(\text{raise}(e)); x.e_2; x.e_3) \} \mapsto \nu \Sigma \{ [e/x]e_3 \} \]

Figure 2: Modal PCF with Exceptions: Dynamics
The exception declaration $\texttt{dclec}\{\tau\}(x.e')$ declares a new exception class, $x$, with associated type $\tau$ for use within $e$. Notice that $x$ is a \textit{variable} that will be replaced by a class reference within the scope $e$; because it is bound within, its name is immaterial. The underlying class name is implicitly bound within $e$, and is not accessible. But, being bound, it, too, is different from any other exception class allocated anywhere in a program.

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 $\texttt{clsfd}$, the $\texttt{exn}$ type. The handle expression $\texttt{handlexc}(e_1; e_2; x.e_3)$ 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. These definitions are given in 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

The derived exception mechanism described here is inspired by the one in Standard ML. It is likely the most widely misunderstood aspect of the language. This mechanism can be separated into two parts, the control aspect and the data aspect. The control aspect is neatly accounted for in modal PCF 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 for the computation type accounts for both of these outcomes in a natural way by providing both a normal continuation and an exceptional continuation.

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 $\tau$ labelled with a dynamically generated class, the class of exception being raised. Exception values are thereby represented as shared secrets between the raiser and the handler. Because the exception class is unguessable, no code, such as library code, can intercept a private exception raised by a client. This is critical to ensuring that modular decomposition of systems remains possible even in the presence of exceptions.\footnote{Contrary to many a methodological claim in the literature attempting to denigrate the use of exceptions.}

Achieving secrecy relies on the computational effect of allocating a dynamically new exception constructor. It is therefore impossible to have a sufficiently strong exception mechanism in a language that separates pure expressions from impure commands. The allocation effect of exception declaration is fundamental, and cannot be avoided. Therefore, languages that lack this capability, such as those that impose a strict separation between expressions and commands, cannot implement exceptions properly.

References
