On Exceptions versus Continuations in the Presence of State

Hayo Thielecke*

ht@dcs.qmw.ac.uk
Department of Computer Science
Queen Mary and Westfield College, University of London
London E1 4NS
UK

Abstract. We compare the expressive power of exceptions and continuations when added to a language with local state in the setting of operational semantics. Continuations are shown to be more expressive than exceptions because they can cause a function call to return more than once, whereas exceptions only allow discarding part of the calling context.

1 Introduction

Exceptions are part of nearly all modern programming languages, including mainstream ones like Java and C++. Continuations are present only in Scheme and the New Jersey dialect of ML, yet are much more intensely studied by theoreticians and logicians. The relationship between exceptions and continuations is not as widely understood as one would hope, partly because continuations, though in some sense canonical, are more powerful than would at first appear, and because the control aspect of exceptions can be obscured by intricacies of typing and syntax.

We have recently shown that exceptions and continuations, when added to a purely functional base language, cannot express each other [11]. That paper affords a comparison of, and contrast between, exceptions and continuations under controlled laboratory conditions, without any contamination from other effects so to speak. In this sequel paper we would like to complete the picture by comparing exceptions and continuations in the presence of state. It is known (and one could call it "folklore") that in the presence of storable procedures, exceptions can be implemented by storing a current handler continuation. It is also plausible that the more advanced uses of continuations cannot be done with exceptions, even if state is available too. Hence we would expect a hierarchy rather than incomparability in the stateful setting.

Formally, we compare expressiveness by means of contextual equivalence. For instance, we showed that $(\lambda x.pxx)M \simeq pMM$ is a contextual equivalence in a language with exceptions, whereas continuations can break it, so that exceptions

* Supported by EPSRC grant GR/L54639

cannot macro-express continuations. Apart from the formal result, we would like to see the equivalences in the stateless setting of [11] as formalizing, at least to some extent, the distinction between the dynamic (exceptions) and the static (continuations) forms of control. The equivalences here give a different perspective, namely of how both forms of control alter the meaning of procedure call. With exceptions, a procedure call may discard part of its calling context; with continuations, a procedure call may return any number of times. It could be said that this distinction reflects the way that control manipulates the call stack: exceptions may erase portions of the stack; continuations may in addition copy them. However, we can make this distinction using only fairly high-level definitions of languages with exceptions and continuations, and a comparison of expressiveness. (Though ideally one would hope for a precise connection between the equivalences that hold for the various forms of control and the demands they put on storage allocation.)

The notion of expressiveness used here was already mentioned by Landin [6, 7], and formalized by Felleisen [3]. The reader should be warned that this notion of expressiveness is very different from the one used by Lillibridge [8]. Lillibridge was concerned with the typing of exceptions in ML, whereas we are concerned only with the actual jumping, that is, raising and handling exceptions, and invoking continuations, respectively. The typing of exceptions in ML "is totally independent of the facility which allows us to raise and handle these wrapped-up objects or packets" [1]. While the language for exceptions used here most closely resembles ML, we do not rely on typing, so that everything is also applicable to the `catch/throw` construct in LISP [14, 13], as it is essentially a spartan exception mechanism without handlers.

The remainder of the paper is organized as follows. The main constructs and their operational semantics are defined in Section 2. We first answer a question from [11], by showing that local exceptions are more powerful than global ones in Section 3. The main result of the paper is that continuations in the presence of state are more powerful than exceptions, which is proved in Section 4. Section 5 sketches how the result here could fit into a more systematic comparison between exceptions and continuations based on how often the current continuation can be used. Section 6 concludes.

2 The Languages and their Operational Semantics

We extend the language used in the companion paper [11] with state by adopting the "state convention" from the Definition of Standard ML [9]. To avoid clutter, the store is elided in the rules unless specified otherwise. Formally a rule

$$\frac{M_1 \Downarrow P_1 \quad \ldots \quad M_n \Downarrow P_n}{M \Downarrow P}$$

is taken to be shorthand for a rule in which the state changes are propagated:

$$\frac{s_0 \vdash M_1 \Downarrow P_1, s_1 \quad \ldots \quad s_{n-1} \vdash M_n \Downarrow P_n, s_n}{s_0 \vdash M \Downarrow P, s_n}$$

Table 1. Natural semantics of the functional subset

$$\frac{P \Downarrow (\lambda x.\ P_1) \quad Q \Downarrow V \quad P_1[x := V] \Downarrow R}{(P\ Q) \Downarrow R} \qquad \frac{N \Downarrow n}{(\texttt{succ}\ N) \Downarrow (n+1)}$$

$$\frac{N \Downarrow 0}{(\texttt{pred}\ N) \Downarrow 0} \qquad \frac{N \Downarrow (n+1)}{(\texttt{pred}\ N) \Downarrow n}$$

$$\frac{N \Downarrow 0 \qquad P_1 \Downarrow R}{(\texttt{if0}\ N\ \texttt{then}\ P_1\ \texttt{else}\ P_2) \Downarrow R} \qquad \frac{N \Downarrow (n+1) \qquad P_2 \Downarrow R}{(\texttt{if0}\ N\ \texttt{then}\ P_1\ \texttt{else}\ P_2) \Downarrow R}$$

$$\frac{}{V \Downarrow V} \qquad \frac{}{(\texttt{rec}\ f(x).\ P) \Downarrow (\lambda x.\ P[f := (\texttt{rec}\ f(x).\ P)])}$$

Table 2. Natural semantics of exceptions

$$\frac{N \Downarrow e \quad P \Downarrow V}{(\texttt{raise}\ N\ P) \Downarrow (\texttt{raise}\ e\ V)} \qquad \frac{N \Downarrow e \quad P \Downarrow V' \quad Q \Downarrow (\texttt{raise}\ e'\ V'') \quad e \neq e'}{(\texttt{handle}\ N\ P\ Q) \Downarrow (\texttt{raise}\ e'\ V'')}$$

$$\frac{N \Downarrow V \quad P \Downarrow V' \quad Q \Downarrow V''}{(\texttt{handle}\ N\ P\ Q) \Downarrow V''} \qquad \frac{N \Downarrow e \quad P \Downarrow V \quad Q \Downarrow (\texttt{raise}\ e\ V') \quad (V\ V') \Downarrow R}{(\texttt{handle}\ N\ P\ Q) \Downarrow R}$$

$$\frac{N \Downarrow (\texttt{raise}\ e\ V)}{(op\ N) \Downarrow (\texttt{raise}\ e\ V)} \qquad \frac{N \Downarrow (\texttt{raise}\ e\ V)}{(\texttt{if0}\ N\ \texttt{then}\ P_1\ \texttt{else}\ P_2) \Downarrow (\texttt{raise}\ e\ V)}$$

$$\frac{P \Downarrow (\texttt{raise}\ e\ V)}{(P\ Q) \Downarrow (\texttt{raise}\ e\ V)} \qquad \frac{P \Downarrow V \quad Q \Downarrow (\texttt{raise}\ e\ V')}{(P\ Q) \Downarrow (\texttt{raise}\ e\ V')}$$

$$\frac{N \Downarrow (\texttt{raise}\ e\ V)}{(\texttt{raise}\ N\ P) \Downarrow (\texttt{raise}\ e\ V)} \qquad \frac{N \Downarrow V \quad P \Downarrow (\texttt{raise}\ e\ V')}{(\texttt{raise}\ N\ P) \Downarrow (\texttt{raise}\ e\ V')}$$

$$\frac{N \Downarrow (\texttt{raise}\ e'\ V)}{(\texttt{handle}\ N\ P\ Q) \Downarrow (\texttt{raise}\ e'\ V)} \qquad \frac{N \Downarrow V \quad P \Downarrow (\texttt{raise}\ e'\ V')}{(\texttt{handle}\ N\ P\ Q) \Downarrow (\texttt{raise}\ e'\ V')}$$

Table 3. Natural semantics of state

$$\frac{s \vdash M \Downarrow a, s_1}{s \vdash (!M) \Downarrow s_1(a), s_1} \qquad \frac{s \vdash M \Downarrow (\texttt{raise}\ e\ V), s_1}{s \vdash (!M) \Downarrow (\texttt{raise}\ e\ V), s_1}$$

$$\frac{s \vdash M \Downarrow V, s_1 \qquad a \notin \mathsf{dom}(s_1)}{s \vdash (\texttt{ref}\ M) \Downarrow a, s_1 + \{a \mapsto V\}} \qquad \frac{s \vdash M \Downarrow (\texttt{raise}\ e\ V), s_1}{s \vdash (\texttt{ref}\ M) \Downarrow (\texttt{raise}\ e\ V), s_1}$$

$$\frac{s \vdash M \Downarrow a, s_1 \qquad s_1 \vdash N \Downarrow V, s_2}{s \vdash (M\texttt{:=}N) \Downarrow V, s_2 + \{a \mapsto V\}}$$

$$\frac{s \vdash M \Downarrow (\texttt{raise}\ e\ V), s_1}{s \vdash (M\texttt{:=}N) \Downarrow (\texttt{raise}\ e\ V), s_1} \qquad \frac{s \vdash M \Downarrow V', s_1 \qquad s_1 \vdash N \Downarrow (\texttt{raise}\ e\ V), s_2}{s \vdash (M\texttt{:=}N) \Downarrow (\texttt{raise}\ e\ V), s_2}$$

Table 4. Evaluation-context semantics of continuations and state

$$
\begin{array}{rcl}
V & ::= & x \mid n \mid a \mid \lambda x.M \mid \texttt{rec}\ f(x).\ M \mid \#E \\
E & ::= & [\cdot] \mid (E\ M) \mid (V\ E) \mid (\texttt{succ}\ E) \mid (\texttt{pred}\ E) \mid (\texttt{if0}\ E\ \texttt{then}\ M\ \texttt{else}\ M) \\
 & \mid & (\texttt{callcc}\ E) \mid (\texttt{throw}\ E\ M) \mid (\texttt{throw}\ V\ E) \\
 & \mid & (\texttt{ref}\ E) \mid (!\ E) \mid (E\texttt{:=}M) \mid (V\texttt{:=}E)
\end{array}
$$

$$
\begin{array}{lll}
s, E[(\lambda x.\ P)\ V] & \to s, E[P[x := V]] & \\
s, E[\texttt{succ}\ n] & \to s, E[n+1] & \\
s, E[\texttt{pred}\ 0] & \to s, E[0] & \\
s, E[\texttt{pred}\ (n+1)] & \to s, E[n] & \\
s, E[\texttt{if0}\ 0\ \texttt{then}\ M\ \texttt{else}\ N] & \to s, E[M] & \\
s, E[\texttt{if0}\ (n+1)\ \texttt{then}\ M\ \texttt{else}\ N] & \to s, E[N] & \\
s, E[\texttt{rec}\ f(x).\ M] & \to s, E[M[f := (\lambda x.\ (\texttt{rec}\ f(x).\ M)\ x)]] & \\
s, E[\texttt{callcc}\ (\lambda x.\ P)] & \to s, E[P[x := (\#E)]] & \\
s, E[\texttt{throw}\ (\#E')\ V] & \to s, E'[V] & \\
s, E[\texttt{ref}\ V] & \to s + \{a \mapsto V\}, E[a] & \text{where } a \notin \mathsf{dom}(s) \\
s, E[!\ a] & \to s, E[s(a)] & \\
s, E[a\texttt{:=}V] & \to s + \{a \mapsto V\}, E[V] &
\end{array}
$$

This version of exceptions (based on the "simple exceptions" of Gunter, Rémy and Riecke [5]) differs from those in ML in that exceptions are not constructors. The fact that exceptions in ML are constructors is relevant chiefly if one does *not* want to raise them, using `exn` only as a universal type. For our purposes, there is no real difference, up to an occasional η-expansion.

Definition 1. *We define the following languages:*

- *Let* λ_V+ *be defined by the operational semantics rules in Table 1.*
- *Let* λ_V+**exn** *be defined by the operational semantics rules in Tables 1 and 2.*
- *Let* λ_V+**state** *be defined by the rules in Table 3 and those in Table 1 subject to the state convention.*
- *Let* λ_V+**exn**+**state** *be defined by the rules in Table 3, as well as those in Tables 1 and 2 subject to the state convention.*

The rules for state are based on those in the Definition of Standard ML [9] (rules (99) and (100) on page 42), except that `ref`, `!` and `:=` are treated as special forms, rather than identifiers. A state is a partial function from addresses to values. For a term M, let $\mathsf{Addr}(M)$ be the set of addresses occurring in M. A program is a closed term P not containing any addresses, that is $\mathsf{Addr}(P) = \emptyset$.

We also need a language with continuations and state:

Definition 2. *Let* λ_V+**cont**+**state** *be the defined by the operational semantics in Table 4.*

The small-step operational semantics of λ_V+**cont**+**state** with evaluation contexts is in the style of Felleisen [12], with store added. Both addresses a and reified continuations $\#E$ are run-time entities that cannot appear in source programs.

Let a context C be a term with a hole not containing addresses.

Definition 3. *Two terms P and P' are* contextually equivalent, *$P \simeq P'$, iff for all contexts C, we have $\emptyset \vdash C[P] \Downarrow n, s$ for some integer n, iff $\emptyset \vdash C[P'] \Downarrow n, s'$.*

Contextual equivalence is defined analogously for the small-step semantics. However, in the small-step semantics we will be concerned with breaking equivalences, a strong version of which is the following:

Definition 4. *Two terms P and P'* can be separated *iff there is a context C such that: $\emptyset, C[P] \rightarrow^* s, n$ for some integer n, and $\emptyset, C[P'] \rightarrow^* s', n'$ with $n \neq n'$.*

(Again, the definition for big-step is analogous.)

Local definitions and sequencing are the usual syntactic sugar:

$$(\texttt{let } x = M \texttt{ in } N) \equiv (\lambda x.N)\, M$$
$$(M; N) \equiv (\lambda x.N)\, M \qquad \text{where } x \text{ is not free in } N$$

3 Local Exceptions Are More Powerful than Global Ones

In this section, we show that even a small amount of state affects our comparison of continuations and exceptions. It may be surprising that local (that is, under a λ) declarations should have state in them, but local exception declarations generate new exception names (somewhat like `gensym` in LISP), and the equality test implicit in the exception handler is enough to make this observable.

Proposition 1. *There are terms that are contextually equivalent in the language with global exceptions λ_V+**exn**, but which can be separated if local exceptions are added.*

Proof. In λ_V+**exn**, we have a contextual equivalence

$$(\lambda x.pxx)\, M \simeq pM\, M$$

The proof of [11, Proposition 1] generalizes to the untyped setting. But local exceptions can break this equivalence: see Figure 1 for a separating context. □

From our perspective, we would maintain that the equivalence holds for the pure control aspect of exceptions, and is broken only because local exceptions are a somewhat hybrid notion with state in them.

Since all we need from local exceptions here is that one term evaluates to 1 and another to 2, we do not give a formal semantics for them, referring the reader to the Definition of Standard ML [9] (for a notation closer to the one used here, see also [5]).

Fig. 1. A separating context using local exceptions in Standard ML

```
fun single m p = let val y = m 0 in p y y end;

fun double m p = p (m 0) (m 0);

fun localnewexn d  =
let
        exception e
        fun r d = raise e
        fun h f x = ((f 0) handle e => x)
in
        fn q => q r h
end;

fun separate copier =
  (copier localnewexn)
    (fn q1 => fn q2 =>
      q1 (fn r1 => fn h1 =>
        q2 (fn r2 => fn h2 =>
          h1 (fn d => h2 r1 1) 2)));

separate single;
val it = 1 : int
separate double;
val it = 2 : int
```

The point in separating (Figure 1) is that each call of `localnewexn` generates a new exception. The handler in `h2` can only handle the exception raised from `r1` if `h2` and `r1` come from the same call of `localnewexn`, as they do in `separate single`, but not in `separate double`.

Local exceptions are relevant for us for two reasons: first, they make the equivalence for exceptions used in [11] inapplicable; second, they can to some extent approximate downward continuations. The example in Figure 1 does perhaps not witness expressive power in any intuitive sense. A more practical example may be the following: can one define a function `f` that passes to some unknown function `g` a function `h` that when called jumps back into `f` (assuming `h` is called before the call of `f` has terminated, because otherwise this would be beyond exceptions). With downward continuations, one can easily do that: in λ_V+**cont**+**state**, we would write `f` as $\lambda g.\texttt{callcc}(\lambda k.g\,(\lambda x.\texttt{throw}\,k\,x))$. Even such pedestrian control constructs as **goto** in ALGOL and `longjmp()` in C could do this. Yet with the simple version of exceptions we have in λ_V+**exn**, a handler in `g` may catch whatever exception `h` wanted to use to jump into `f`. With local exceptions however, `f` could declare a local exception for `h` to raise, which would thus be distinct from any that `g` could handle. On the other hand, language designers specifically chose to equip `g` so that it can intercept jumps from `h` to

`f`: in ML even local exceptions can be handled by using a variable (or just a wildcard) pattern in the handler, while LISP provides `unwind-protect`.

4 Exceptions Cannot Make Functions Return Twice

Encodings of exceptions in terms of stored continuations have been known for some time, and can probably be regarded as folklore [5]; see also Reynolds's textbook [10]. It would still be worthwhile to analyze encodings of the various notions of exceptions in more detail. But the fact that such an encoding is possible, and that consequently continuations and state are *at least as* expressive as exceptions and state, will be treated as a known result here. We will strengthen it by showing that continuations in the presence of state are *strictly more* expressive than exceptions.

Define terms R_1 and R_2 in λ_V+**state** by

$$R_j \equiv \lambda z.((\lambda x.\lambda y.(z\,0;\ x\texttt{:=}\ !y;\ y\texttt{:=}j;\ !x))\,(\texttt{ref}\,0)\,(\texttt{ref}\,0))$$

Informally, the idea is that j is hidden inside R_j. As the variables x and y are local, the only way to observe j would be to run the assignments after the call to z twice, so that j is first moved into y, and then x, whose value is returned at the end. With exceptions, that is impossible.

The proof uses a variant of the technique used for exceptions in [11], extended to deal with the store. First we define a relation needed for the induction hypothesis:

Definition 5. *We define relations $\sim$ and $\sim_A$, where A is a set of addresses, as follows:*

- *on terms, let $\sim$ be the least congruence such that $M \sim M$ and $R_j \sim R_{j'}$ for any integers j and j';*
- *for stores, let $s \sim_A s'$ iff $A \subseteq \mathsf{dom}(s) = \mathsf{dom}(s')$ and for all $a \in A$, $s(a) \sim s'(a)$ and $\mathsf{Addr}(s(a)) \subseteq A$;*
- *for stores together with terms, let $s, M \sim_A s', M'$ iff $s \sim_A s'$ and $M \sim M'$, and also $\mathsf{Addr}(M) \subseteq A$.*

Intuitively, $s, M \sim_A s', M'$ implies that M in store s and M' in store s' are linked in lockstep; but the stores may differ in addresses outside A, which are inaccessible from M.

Lemma 1. *Assume $s, P \sim_A s', P'$ and $s \vdash P \Downarrow Q, s_1$. Then there exist a term Q', a store s'_1 and a set of addresses A_1 such that*

- *$s' \vdash P' \Downarrow Q', s'_1$;*
- *$s_1, Q \sim_{A_1} s'_1, Q'$;*
- *$A \subseteq A_1$ and $(\mathsf{dom}(s) \setminus A) \subseteq (\mathsf{dom}(s_1) \setminus A_1)$;*
- *for all addresses $a \in \mathsf{dom}(s) \setminus A$, the stores satisfy $s_1(a) = s(a)$ and $s'_1(a) = s'(a)$.*

Proof. Proof by induction on the derivation of $s \vdash P \Downarrow Q, s_1$. We assume $s, P \sim_A s', P'$ and proceed by cases on the last rule applied in the derivation.

Case $P \equiv MN$ and $s \vdash MN \Downarrow Q, s_4$. The last rule is

$$\frac{s \vdash M \Downarrow \lambda z.M_1, s_1 \qquad s_1 \vdash N \Downarrow V_2, s_2 \qquad s_2 \vdash M_1[z := V_2] \Downarrow Q, s_4}{s \vdash M\ N \Downarrow Q, s_4}$$

As $MN = P \sim P'$, P' must be of the form $M'N'$. By the induction hypothesis applied to $s \vdash M \Downarrow (\lambda z.M_1), s_1$, we have $s' \vdash M' \Downarrow (\lambda z.M_1'), s_1'$, with $s_1, \lambda z.M_1 \sim_{A_1} s_1', \lambda z.M_1'$.
There are two possible cases implied by $\lambda z.M_1 \sim \lambda z.M_1'$: either $M_1 \sim M_1'$; or $\lambda z.M_1 = R_j$ and $\lambda z.M_1' = R_{j'}$. In the first case, the claim follows by repeatedly applying the induction hypothesis. So suppose the second, that $\lambda z.M_1 = R_j$ and $\lambda z.M_1' = R_{j'}$. We apply the induction hypothesis, giving us $s_1', N' \Downarrow V_2', s_2'$ with $s_2, V_2 \sim_{A_2} s_2', V_2'$. Now

$$M_1[z := V_2] = (\lambda x.\lambda y.(V_2\, 0;\ x\texttt{:=}\ !y;\ y\texttt{:=}j;\ !x))\ (\texttt{ref}\ 0)\ (\texttt{ref}\ 0)$$

This term will allocate two new addresses, so let $a, b \notin \mathsf{dom}(s_2)$. Then $s_2 \vdash M_1[z := V_2] \Downarrow Q, s_4$ iff

$$s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0; a\texttt{:=}\,!b; b\texttt{:=}j; !a \Downarrow Q, s_4$$

There are two possible cases, depending on whether $V_2\, 0$ in store $s_2 + \{a \mapsto 0, b \mapsto 0\}$ raises an exception or not. First, suppose it does, that is,

$$s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0 \Downarrow \texttt{raise}\ e\ V_3, s_3 \tag{1}$$

As $s_2 + \{a \mapsto 0, b \mapsto 0\}, V_2\, 0 \sim_{A_2} s_2 + \{a \mapsto 0, b \mapsto 0\}, V_2'\, 0$, the induction hypothesis implies

$$s_2' + \{a \mapsto 0, b \mapsto 0\} \vdash V_2'\, 0 \Downarrow \texttt{raise}\ e\ V_3', s_3'$$

with $\texttt{raise}\ e\ V_3, s_3 \sim_{A_2} \texttt{raise}\ e\ V_3', s_3'$. The exception propagates, devouring the difference between j and j' in this call of R_j, more technically:

$$\frac{\frac{\frac{\frac{s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0 \Downarrow \texttt{raise}\ e\ V_3, s_3}{s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0; a\texttt{:=}\,!b \Downarrow \texttt{raise}\ e\ V_3, s_3}}{s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0; a\texttt{:=}\,!b; b\texttt{:=}j \Downarrow \texttt{raise}\ e\ V_3, s_3}}{s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0; a\texttt{:=}\,!b; b\texttt{:=}j; !a \Downarrow \texttt{raise}\ e\ V_3, s_3}$$

That is, $s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash M_1[z := V_2] \Downarrow \texttt{raise}\ e\ V_3, s_3$, hence the whole call raises an exception

$$s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash MN \Downarrow \texttt{raise}\ e\ V_3, s_3 \tag{2}$$

Analogously for V_2'. Letting $Q = \texttt{raise}\ e\ V_3$ and $s_4 = s_3$, we are done for this subcase. Now assume $V_2\, 0$ does not raise an exception, so that there is a value V_3 returned by the call:

$$s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0 \Downarrow V_3, s_3 \tag{3}$$

We apply the induction hypothesis to the call $V_2\,0$, relying on the fact that V_2 can only reach addresses in A_2, so that it cannot modify the newly allocated a and b:

$$s_2 + \{a \mapsto 0, b \mapsto 0\}, V_2\,0 \sim_{A_2} s_2' + \{a \mapsto 0, b \mapsto 0\}, V_2'\,0$$

The induction hypothesis thus gives us $s_2' + \{a \mapsto 0, b \mapsto 0\} \vdash V_2'\,0 \Downarrow V_3', s_3'$, and $s_3, V_3 \sim_{A_3} s_3', V_3'$. As $b \in \mathsf{dom}(s_2 + \{a \mapsto 0, b \mapsto 0\})$, but $b \notin A_2$, we have $s_3(b) = 0$, and $s_3'(b) = 0$, and also $b \notin A_3$. Putting the pieces together, we derive:

$$s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash (V_2\,0;\ a\texttt{:=}\ !b;\ b\texttt{:=}j;\ !a) \Downarrow 0, s_3 + \{b \mapsto j\}$$

hence

$$s_2 + \{a \mapsto 0, b \mapsto 0\}, (\lambda x.\lambda y.(V_2\,0;\ x\texttt{:=}\ !y;\ y\texttt{:=}j;\ !x))\ (\texttt{ref}\,0)\ (\texttt{ref}\,0) \Downarrow 0, s_3 + \{b \mapsto j\}$$

Analogously

$$s_2' + \{a \mapsto 0, b \mapsto 0\} \vdash (V_2'\,0;\ a\texttt{:=}\ !b;\ b\texttt{:=}j';\ !a) \Downarrow 0, s_3' + \{b \mapsto j'\}$$

hence

$$s_2' + \{a \mapsto 0, b \mapsto 0\}, (\lambda x.\lambda y.(V_2'\,0;\ x\texttt{:=}\ !y;\ y\texttt{:=}j';\ !x))\ (\texttt{ref}\,0)\ (\texttt{ref}\,0) \Downarrow 0, s_3' + \{b \mapsto j'\}$$

Thus

$$s \vdash MN \Downarrow 0, s_3 + \{b \mapsto j\} \tag{4}$$

and $s' \vdash M'N' \Downarrow 0, s_3 + \{b \mapsto j'\}$ with $s_3 + \{b \mapsto j\}, 0 \sim_{A_3} s_3' + \{b \mapsto j'\}, 0$, as required. This is the linchpin of the whole proof: b holds j or j', respectively; but that is of no consequence, because b, lying outside of A_3, is garbage.

Case $P \equiv !M$ **and** $s \vdash !M \Downarrow s_1(a), s_1$**.** Hence $s \vdash M \Downarrow a, s_1$. As $!M \sim P'$, P' must be of the form $!M'$ with $M \sim M'$. By the induction hypothesis, $s' \vdash M' \Downarrow Q', s_1'$ with $s_1, a \sim_{A_1} s_1', Q'$, and $\mathsf{Addr}(s(a)) \subseteq A_1$. As this implies $a \sim Q'$, we have $a = Q'$, so that $s' \vdash M' \Downarrow a, s_1'$, which implies $s' \vdash !M' \Downarrow s_1'(a), s_1'$. As $a = \mathsf{Addr}(Q') \subseteq A_1$, $s_1(a) \sim s_1'(a)$. Thus $s_1, s_1(a) \sim_{A_1} s_1', s_1'(a)$, as required.

Case $P \equiv \texttt{ref}\,M$ **and** $s \vdash \texttt{ref}\,M \Downarrow a, s_1 + \{a \mapsto V\}$**.** Hence $s \vdash M \Downarrow V, s_1$ with $a \notin \mathsf{dom}(s_1)$. As $\texttt{ref}\,M \sim P'$, P' must be of the form $\texttt{ref}\,M'$ with $M \sim M'$. By the induction hypothesis, $s' \vdash M' \Downarrow V', s_1'$, with $s_1, V \sim_{A_1} s_1', V'$ where $A \subseteq A_1$. Thus $s' \vdash \texttt{ref}\,M' \Downarrow a, s_1' + \{a \mapsto V'\}$. (We can pick the same a, because $a \notin \mathsf{dom}(s_1') = \mathsf{dom}(s_1)$.) Thus, $s' \vdash \texttt{ref}\,M' \Downarrow a, s_1' + \{a \mapsto V'\}$ with

$$s_1 + \{a \mapsto V\}, a \sim_{A_1 \cup \{a\}} s_1' + \{a \mapsto V'\}, a$$

Furthermore, $A \subseteq A_1 \cup \{a\}$ and $\mathsf{dom}(s) \setminus A \subseteq \mathsf{dom}(s_1 + \{a \mapsto V\}) \setminus (A_1 \cup \{a\})$.

Case $P \equiv (M := N)$ **and** $s \vdash M := N \Downarrow V, s_2 + \{a \mapsto V\}$**.** Hence $s \vdash M \Downarrow a, s_1$ and $s_1 \vdash N \Downarrow V, s_2$. As $M := N \sim P'$, P' must be of the form $M' := N'$, with $M \sim M'$ and $N \sim N'$. Applying the induction hypothesis to $s \vdash M \Downarrow a, s_1$ gives us Q' and s_1' such that $s' \vdash M' \Downarrow Q', s_1'$ and $s_1, a \sim_{A_1} s_1', Q'$. So $a \sim Q'$, which means $Q' = a$. Applying the induction hypothesis to $s_1 \vdash N \Downarrow V, s_2$ and $s_1, N \sim_{A_1} s_1', N'$ gives us V' and s_2' such that $s_2, V \sim_{A_2} s_2', V'$. Thus $s' \vdash M' := N' \Downarrow V', s_2' + \{a \mapsto V'\}$ with

$$s_2 + \{a \mapsto V\}, V \sim_{A_2} s_2' + \{a \mapsto V'\}, V'$$

as required. Assume b is an address with $b \in \mathsf{dom}(s) \setminus A$. Then $b \notin A_2$, and $s_2(b) = s_1(b) = s(b)$. Because $a \in A_2$, we have $b \neq a$, so that the store $s_2 + \{a \mapsto V\}$ still maps b to $s(b)$.

Otherwise. The last rule in the derivation must be of the form

$$\frac{s \vdash P_1 \Downarrow Q_1, s_1 \quad \ldots \quad s_{n-1} \vdash P_n \Downarrow Q_n, s_n}{s \vdash P \Downarrow Q, s_n}$$

Observe that the P_i in the antecedents are the immediate subterms of the P in the conclusion, and that conversely the Q in the conclusion is assembled from subterms of P and some of the Q_i in the antecedents. Hence:

$$\mathsf{Addr}(P_1) \cup \ldots \cup \mathsf{Addr}(P_n) \subseteq \mathsf{Addr}(P)$$
$$\mathsf{Addr}(Q) \subseteq \mathsf{Addr}(P) \cup \mathsf{Addr}(Q_1) \cup \ldots \cup \mathsf{Addr}(Q_n)$$

Because $s, P \sim_A s', P'$, we have $s \sim s'$ and $P \sim P'$. The case $P \equiv R_j$ is trivial; otherwise we have $P \sim P'$ due to congruence, so there are $P_1', \ldots, P_n'$ with $P_i \sim P_i'$. Now $s, P_1 \sim_A s', P_1'$ (because $\mathsf{Addr}(P_1) \subseteq \mathsf{Addr}(P) \subseteq A$). By the induction hypothesis, there exist Q_1', s_1' and A_1 such that $s_1, Q_1 \sim_{A_1} s_1', Q_1'$ and $A \subseteq A_1$. Hence $s_1, P_2 \sim_{A_1} s_1', P_2'$, so that we can apply the induction hypothesis again to $s_1 \vdash P_2 \Downarrow Q_2, s_2$, and so on for all the antecedents.
Finally, let Q' be built up from the Q_i' in the same way as Q is built up from the Q_i. By congruence, we have $Q \sim Q'$. As $s_n \sim s_n'$ and $\mathsf{Addr}(Q) \subseteq A_n$, we have $s_n, Q \sim_{A_n} s_n', Q'$, as required. □

We have thus shown that terms containing R_1 and R_2, respectively, proceed in lockstep. This implies that the R_j are contextually equivalent:

Lemma 2. *R_1 and R_2 are contextually equivalent in λ_V+**exn**+**state**.*

Proof. Let C be a context. Suppose $\emptyset \vdash C[R_1] \Downarrow n, s$ for some integer n. We need to show that $C[R_2]$ also reduces to n. First, note that because $\sim$ on terms is defined to be a congruence with $R_1 \sim R_2$, we have $C[R_1] \sim C[R_2]$. As neither of these terms contains any addresses, they are related in the empty store with respect to the empty set of addresses, that is $\emptyset, C[R_1] \sim_\emptyset \emptyset, C[R_2]$. By Lemma 1, we have $\emptyset \vdash C[R_2] \Downarrow Q', s'$, for some s', Q' and A such that $s, n \sim_A s', Q'$. This implies $n \sim Q'$, so that $n = Q'$. The argument for showing that $\emptyset \vdash C[R_2] \Downarrow n, s$ implies that $C[R_1]$ in the empty store also reduces to n is symmetric. □

Fig. 2. A separating context using continuations and state in SML/NJ

```
fun R j z = (fn x => fn y => (z 0; x := !y; y := j; !x))(ref 0)(ref 0);

fun C Rj =
callcc(fn top =>
        let
            val c = ref 0
            val s = ref top
            val d = Rj (fn p => callcc(fn r => (s := r; 0)))

        in
            (c := !c + 1;
             if !c = 2 then d else throw (!s) 0)
        end);

C(R 1);
val it = 1 : int
C(R 2);
val it = 2 : int
```

Note that the proof would still go through if we changed the notion of observation to termination, or if we restricted to the typed subset.

It remains to show that the two terms that are indistinguishable with exceptions and state can be separated with continuations and state. To separate, the argument to R_j should save its continuation, then restart that continuation once, so the assignments get evaluated twice, thereby assigning j to x, and thus making the concealed j visible to the context.

Lemma 3. *In* λ_V+**cont**+**state**, R_1 *and* R_2 *can be separated: there is a context* $C[\cdot]$ *such that*

$$\emptyset, C[R_1] \to^* s_1, 1$$
$$\emptyset, C[R_2] \to^* s_1', 2$$

This is actually strictly stronger than R_1 and R_2 not being contextually equivalent (and it is machine-checkable by evaluation). We omit the lengthy calculation here, but see Figure 2 for the separating context written in Standard ML of New Jersey. From Lemmas 2 and 3, we conclude our main result:

Proposition 2. *There are* λ_V+**state** *terms that are contextually equivalent in* λ_V+**exn**+**state**, *but which can be separated in* λ_V+**cont**+**state**.

Combined with the known encodings of exceptions in terms of continuations and state, Proposition 2 means that continuations in the presence of state are *strictly more* expressive than exceptions.

5 Exceptions Can Discard the Calling Context

We have established that continuations are more expressive than exceptions by showing how they affect functions calls: using continuations, a call can return more than once. In this section, we aim at an analogous result for showing how exceptions give rise to added power compared to a language without control: using exceptions, a function call may discard part of the calling context. To put it facetiously as a contest between a term and its context, in the previous section we concocted a calling context whose main ingredient

$$\ldots z\,0; x := !\,y; y := j; !\,x \ldots$$

was chosen such that something good (for separating) would happen if only the callee z could return twice. Now we need a calling context in which something bad happens if the callee returns at all. One such context is given by sequencing with divergence. (The callee could avoid ever returning to the divergence by diverging itself, but for separating that would defeat the purpose.) More formally, there are terms that are contextually equivalent in the language with state but no control, and which can be separated in the language with exceptions and state (in fact, in any language with control). Let Ω be the diverging term $((\mathtt{rec}\ f(x).\ f\,x)\,0)$. The recursion construct is used here so that everything generalizes to the typed subset of λ_V+**state**; if we are only concerned with the untyped language, we could just as well put $\Omega = (\lambda x.xx)(\lambda x.xx)$. Analogously to Lemma 2, we have

Lemma 4. *$(M;\Omega)$ and $(N;\Omega)$ are contextually equivalent in λ_V+**state**.*

Proof. (Sketch) Let $\sim$ be the least congruence such that $M \sim M$ and $M;\Omega \sim N;\Omega$ for any M and N. Let $\sim$ be defined on states pointwise, and let $s, P \sim s', P'$ iff $s \sim s'$ and $P \sim P'$. As in Lemma 1, we need to show that if $s, P \sim s', P'$ and $s \vdash P \Downarrow Q, s_1$, there is a Q such that $s' \vdash P' \Downarrow Q', s_1'$ with $s_1, Q \sim s_1', Q'$. The only non-trivial case if $P \equiv (M;\Omega)$ and $P' \equiv (N;\Omega)$. Suppose one of them reduces to integer. If we do not have control constructs, that can only be the case if Ω reduces to a value. But here is no V such that $s, \Omega \Downarrow V, s_1$. (For suppose they were: there would be a derivation of minimal height, which would have to contain a smaller one.) □

The proof is simpler than for exceptions because when we relate two terms $(M;\Omega)$ and $(N;\Omega)$ it does not matter what M and N do, or what storage they allocate, as the Ω prevents any observation.

Lemma 5. *$(M;\Omega)$ and $(N;\Omega)$ can be separated in λ_V+**exn**+**state**.*

Proof. Let

$$\begin{aligned} M &= \mathtt{raise}\ e\ 1 \\ N &= \mathtt{raise}\ e\ 2 \\ C &= \mathtt{handle}\ e\ [\cdot]\ (\lambda x.x) \end{aligned}$$

Then we have $\emptyset \vdash C[M;\Omega] \Downarrow 1, \emptyset$ and $\emptyset \vdash C[N;\Omega] \Downarrow 2, \emptyset$ in λ_V+**exn**+**state**. □

Proposition 3. *There are two terms in λ_V+ that are contextually equivalent in λ_V+**state**, but which can be separated in λ_V+**exn**+**state**.*

So far we have used operational semantics and contextual equivalence as a kind of probe to observe what control constructs can and cannot do. The astute reader may however have begun to suspect what the preoccupation with discarding the current continuation, or using it more than once, is driving at. In the remainder of this section, we sketch how the earlier material fits in with *linearity* in the setting of continuation semantics.

It is evident that in the continuation semantics of a language without control operators the current continuation is used in a linear way. For the function type we have

$$[\![A \to B]\!] = ([\![B]\!] \to \mathsf{Ans}) \multimap ([\![A]\!] \to \mathsf{Ans})$$

In a language with `callcc`, the $\multimap$ would have to be replaced by a $\to$, because the current continuation could be discarded or copied. Domain-theoretically, the linear arrow $\multimap$ can be interpreted as strict function space. So in the case of $M; \Omega$, the meaning of a looping term is $[\![\Omega]\!] = \bot$, and because

$$[\![M]\!] : \mathsf{Env} \to ([\![B]\!] \to \mathsf{Ans}) \multimap \mathsf{Ans}$$

is strict in its continuation argument, it preserves $\bot$. So it is immediate that

$$[\![M; \Omega]\!] \;= \bot = \; [\![N; \Omega]\!]$$

Moreover, this argument is robust in the sense that it works the same in the presence of state. In the semantics of a language with state, expression continuations take the store as an additional argument, so that the meaning of M is now:

$$[\![M]\!] : \mathsf{Env} \to \mathsf{Store} \to ([\![B]\!] \to \mathsf{Store} \to \mathsf{Ans}) \multimap \mathsf{Ans}$$

This is still strict in its continuation argument, mapping the divergent continuation $\bot$ to $\bot$.

All this requires little more than linear typechecking of the CPS transform. What seems encouraging, however, is that exceptions begin to fit into the same framework. For a language with exceptions or dynamic `catch`, the continuation semantics passes a current handler continuation. Here the current continuation and the handler continuation *together* are subject to linearity (this linearity is joint work in progress with Peter O'Hearn and Uday Reddy, which may appear elsewhere). Assuming that all exceptions are injected into some type E (like `exn` in ML), the linearity is seen most clearly in the function type:

$$[\![A \to B]\!] = (([\![B]\!] \to \mathsf{Ans}) \& ([\![E]\!] \to \mathsf{Ans})) \multimap ([\![A]\!] \to \mathsf{Ans})$$

(Note that this linear use of non-linear continuations is quite different from "linear continuations" [4]). Again the linearity would remain the same if state were added to the continuations. The current continuation can be discarded in favour of the handler, but never used twice. Exceptions thus occupy a middle

ground between no control operators (*linear* usage of the current continuation) and first-class continuations (*intuitionistic*, that is unrestricted, usage). For this reason we regard Lemma 2, which confirms that with exceptions no function call can return twice, as more than a random equivalence: it seems to point towards deeper structural properties of control made observable by the presence of state (in that the `ref` construct allowed us to "stamp" continuations uniquely, and then to count their usage with assignments).

6 Conclusions and Directions for Further Research

It is striking how sensitive the comparison between exceptions and continuations is to the chosen measure of expressiveness: in Lillibridge's terms, "exceptions are strictly more powerful" than continuations [8]; in terms of contextual equivalence and in the absence of state they are incomparable [11]; while in the presence of state, continuations are strictly more expressive than exceptions. The last of these is perhaps the least surprising because closest to programming intuition.

Each of these notions is to some extent brittle. For instance, comparisons of expressiveness based on the ability to encode recursion are inapplicable if the language under consideration already has recursion—and in the presence of state (including storable procedures) that is inevitable, as one can use Landin's technique of "tying a knot in the store" to define the "imperative **Y**-combinator". On the other hand, the technique of witnessing expressive power by breaking equivalences, while more widely applicable, is not completely robust either, if other effects are added to the language that already break the equivalence: compare Proposition 1 and [11, Proposition 1]. (Equivalences may be broken for uninteresting as well as interesting reasons.) Furthermore, while we would claim that Proposition 2 confirms and backs up programming intuition, it can hardly be said to *express* the difference between exceptions and continuations. A type system for the restricted (linear or affine) use of the current continuation would come much closer to achieving this. Ideally, such a linear typing for continuation-passing style together with typed equivalences of the target language should entail the equivalences considered here; we hope that our results will give such a unified treatment something to aim for. It has been suggested to us that "of course exceptions are weaker—they're on the stack". Some substance might conceivably be added to such statements if it could be shown that linearity in the use of continuations by dynamic control constructs is what allows control information to be stack-allocated (see also [2, 15]).

Acknowledgements Thanks to Jon Riecke, Peter O'Hearn and Josh Berdine.

References

1. Andrew Appel, David MacQueen, Robin Milner, and Mads Tofte. Unifying exceptions with constructors in Standard ML. Technical Report ECS LFCS 88 55,

Laboratory for Foundations of Computer Science, University of Edinburgh, June 1988.
2. Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. *ACM SIGPLAN Notices*, 31(5):99–107, May 1996.
3. Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, volume 17, pages 35–75, 1991.
4. Andrzej Filinski. Linear continuations. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, 1992.
5. Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 12–23, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.
6. Peter J. Landin. A generalization of jumps and labels. Report, UNIVAC Systems Programming Research, August 1965.
7. Peter J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2), 1998. Reprint of [6].
8. Mark Lillibridge. Exceptions are strictly more powerful than Call/CC. Technical Report CMU-CS-95-178, Carnegie Mellon University, July 1995.
9. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
10. John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.
11. Jon G. Riecke and Hayo Thielecke. Typed exceptions and continuations cannot macro-express each other. In Jǐrí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Procedings ICALP '99*, volume 1644 of *LNCS*, pages 635–644. Springer Verlag, 1999.
12. Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: full abstraction for models of control. In M. Wand, editor, *Lisp and Functional Programming*. ACM, 1990.
13. Guy L. Steele. *Common Lisp: the Language*. Digital Press, 1990.
14. Guy L. Steele and Richard P. Gabriel. The evolution of Lisp. In Richard L. Wexelblat, editor, *Proceedings of the Conference on History of Programming Languages*, volume 28(3) of *ACM Sigplan Notices*, pages 231–270, New York, NY, USA, April 1993. ACM Press.
15. Mitchell Wand and Dino P. Oliva. Proving the correctness of storage representations. In *1992 ACM Conferenc on Lisp and Functional Programming*, pages 151–160. ACM, ACM, August 1992.