Exceptions, Continuations and Macro-expressiveness

James Laird

COGS, University of Sussex, UK
e-mail: jiml@cogs.susx.ac.uk

Abstract. This paper studies the the problem of expressing exceptions using first-class continuations in a functional-imperative language. The main result is that exceptions *cannot* be macro-expressed using first-class continuations and references (contrary to "folklore"). This is shown using two kinds of counterexample. The first consists of two terms which are equivalent with respect to contexts containing continuations and references, but which can be distinguished using exceptions. It is shown, however, that there are no such terms which do not contain `callcc`. However, there is a Π_1 sentence of first-order logic which is satisfied when interpreted in the domain of programs containing continuations and references but not satisfied in the domain of programs with exceptions and references. This is used to show that even when `callcc` is omitted from the source language, exceptions still cannot be expressed using continuations and references.

1 Introduction

All practical functional programming languages have operators for manipulating the flow of control, typically either first-class *continuations* or *exceptions*. There are clear differences between these features; the former are *statically* scoped, whilst the latter are handled *dynamically*. But how significant are these distinctions? (After all, many instances of control can be written using either continuations or exceptions.) This can be seen as a question of *relative expressive power*: can exceptions be expressed in terms of continuations and vice-versa [3]? The difficulty lies in formalising this problem; there is a consensus that translations between languages should be the basis for comparing their expressive power but different notions of what constitutes a satisfactory translation have been proposed in different contexts [11,7,1]. Having settled on one of them, a second problem is that translations compare whole languages rather than specific features, as Felleisen notes [1]:

> ... claims [about expressiveness] only tend to be true in specific language universes for specific conservative language restrictions: they often have no validity in other contexts!

For example, Lillibridge [6] has shown that adding ML-style exceptions to the simply-typed λ-calculus allows recursion to be encoded whilst adding `callcc`

D. Le Métayer (Ed.): ESOP 2002, LNCS 2305, pp. 133–146, 2002.

does not, and so by this measure exceptions are more powerful than continuations. However, "realistic" languages include some form of recursion directly and so this notion of expressiveness based on computational strength is too coarse to distinguish exceptions from `callcc` in general.

A more fine-grained approach is obtained if the translations between languages are restricted to those which preserve common structure — the terms themselves and their contextual equivalences — intact. Riecke and Thielecke [12] have shown that in the context of a simple functional language, exceptions cannot be expressed as a macro in terms of continuations and vice-versa, by giving terms of the simply-typed λ-calculus which are contextually equivalent when continuations are added to the language but inequivalent if exceptions are added, and terms which are equivalent in the presence of exceptions but not continuations. This does not resolve the issue, however, for languages like ML or Scheme which combine exceptions or continuations with assignable *references*. Thielecke [14] has shown that in the presence of state it is still the case that exception-handling cannot be used to macro-express `callcc`. He also observes that "...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." (An example of such an implementation is given in [10].) The inference drawn from this fact in [14] is that in the presence of higher-order references, continuations are strictly more expressive than exceptions, although, as we shall see, the implementation *fails* to conform to the criteria given in [1] and adopted in [12,14] for a translation to be used to compare expressiveness.

1.1 Contribution of This Paper

The main formal results contained in this paper are two counterexamples to the assertion that exceptions can be expressed in terms of continuations and references. (An appendix gives (Standard ML of New Jersey) code corresponding to these examples.) Section 2 describes a language with exceptions, continuations and references, and gives some background on the theory of expressivenes of programming languages. The first counterexample (Section 3) is of the kind used in [12,14]; two terms which are observationally equivalent in a language with continuations and references but which can be distinguished when exceptions are added. In fact, these two terms are actually equated by various "theories of control" such as the $\lambda\mathcal{C}$ [2] and $\lambda\mu$-calculi [9], and hence these theories *are not sound* in the presence of exceptions.

However, the terms used in the first counterexample contain `callcc`, so it does not exclude the possibility that exceptions (in the absence of continuations) can be reduced to continuations and references. In fact, it is shown in Section 4 that no counterexample of the same form can exclude this possibility — because the implementation of exceptions using `callcc` and references preserves equivalences between terms which don't contain `callcc`.

Hence to show that the λ-calculus with exceptions and references cannot be reduced to the λ-calculus with continuations and references (Section 5), it proves necessary to develop the theory of expressiveness by describing a new and more

general kind of counterexample to relative expressiveness — a Π_1 formula of the associated logic of programs which is satisfied in the domain of programs containing continuations and references, but not satisfied when programs can contain exceptions.

2 Exceptions, Continuations and References

Following [1], a programming language $\mathcal{L}$ will be formally considered to be a tuple $\langle \mathrm{Tm}_{\mathcal{L}}, \mathrm{Prog}_{\mathcal{L}} \subseteq_{\mathcal{L}}, \mathrm{Eval}_{\mathcal{L}} \rangle$ consisting of the terms of $\mathcal{L}$ (phrases generated from a signature Σ_L — a set of constructors with arities), the well-formed programs of $\mathcal{L}$ and the terminating programs of $\mathcal{L}$ respectively ($M \in \mathrm{Eval}_L$ will be written $M \Downarrow_{\mathcal{L}}$). $\mathcal{L}'$ is a conservative extension of $\mathcal{L}$ ($\mathcal{L} \subseteq \mathcal{L}'$) if $\mathrm{Tm}_{\mathcal{L}} \subseteq \mathrm{Tm}_{\mathcal{L}'}$, $\mathrm{Prog}_{\mathcal{L}} = \mathrm{Prog}_{\mathcal{L}'} \cap \mathrm{Tm}_{\mathcal{L}}$ and $\mathrm{Eval}_{\mathcal{L}} = \mathrm{Eval}_{\mathcal{L}'} \cap \mathrm{Prog}_{\mathcal{L}}$.

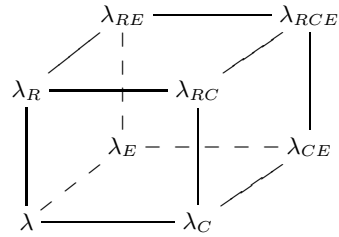

Fig. 1. A hierarchy of functional languages with side-effects

The languages considered here will all be contained within λ_{RCE}, a λ-calculus extended with exceptions, continuations and references. The various fragments of λ_{RCE} — λ-calculus with just references, with just exceptions, with just continuations, with exceptions and references, and with continuations and references — will be referred to as λ_R, λ_E, λ_C, λ_{RE} and λ_{RC} respectively (Figure 1).
Terms of λ_{RCE} are formed according to the following grammar:

$M ::= x \mid * \mid \lambda x.M \mid M\ M \mid$
$\quad$ `abort` $M \mid$ `callcc` $M \mid$
$\quad$ `new_exn` $\mid$ `raise` $M \mid$ `handle` $M\ M \mid$
$\quad$ `new` $\mid M := M \mid !M.$

We shall consider a typed form of λ_{RCE}, although the results described here apply (in a modified form) in an untyped setting as well. Types are generated from a basis — including a type `unit` containing the single value $*$, an empty type $\mathbf{0}$, and a type `exn` of exceptions — by $\Rightarrow$ and a type constructors for references:

$T ::= \mathtt{unit} \mid \mathbf{0} \mid \mathtt{exn} \mid T \Rightarrow T \mid T\ \mathtt{ref}$

Terms are derived in contexts of typed variables; the typing judgements for exceptions, references and continuations are as follows:

$$\frac{}{\vdash \mathtt{new}_T : T\ \mathtt{ref}} \qquad \frac{\Gamma \vdash M : T\ \mathtt{ref} \quad \Gamma \vdash N : T}{\Gamma \vdash M := N : \mathtt{unit}} \qquad \frac{\Gamma \vdash M : T\ \mathtt{ref}}{\Gamma \vdash !M : T}$$

$$\frac{}{\vdash \mathtt{new\_exn} : \mathtt{exn}} \qquad \frac{\Gamma \vdash M : \mathtt{exn}}{\Gamma \vdash \mathtt{raise}\, M : S} \qquad \frac{\Gamma \vdash M : \mathtt{exn} \quad \Gamma \vdash N : \mathtt{unit}}{\Gamma \vdash \mathtt{handle}\, M\, N : \mathtt{unit}}$$

$$\frac{\Gamma \vdash M : \mathbf{0}}{\Gamma \vdash \mathtt{abort}\, M : T} \qquad \frac{\Gamma \vdash M : (T \Rightarrow S) \Rightarrow T}{\Gamma \vdash \mathtt{callcc}\, M : T}$$

Reference and exception variables have been given the local declarations `new` and `new_exn` (although the counterexample of Section 3 also shows that global exceptions cannot be expressed by continuations and references). *Evaluation contexts* are used to pick out the next redex, and to represent continuations.

Definition 1. *Evaluation contexts are given by the following grammar:*
$E[\cdot] ::= [\cdot] \mid E[\cdot]\, M \mid V\, E[\cdot] \mid$
$\quad \mathtt{callcc}\, E[\cdot] \mid$
$\quad E[\cdot] := M \mid V := E[\cdot] \mid\ !E[\cdot] \mid$
$\quad \mathtt{raise}\, E[\cdot] \mid \mathtt{handle}\, E[\cdot]\, M \mid \mathtt{handle}\, h\, E[\cdot]$
where M ranges over general terms, and V over values (variables, exception and reference names, and lambda abstractions).

Our exceptions are essentially a simplified version of Gunther, Rémy and Riecke's "simple exceptions" [3]. A modified notion of evaluation context is used to determine which handler will trap an exception; for each exception name h, $E_h[\cdot]$ ranges over the evaluation contexts which do not have a $\mathtt{handle}\, h$ in their spine.
$E[\cdot] ::= [\cdot] \mid E_h[\cdot]\, M \mid V\, E_h[\cdot] \mid$
$\quad \mathtt{callcc}\, E_h[\cdot] \mid$
$\quad E_h[\cdot] := M \mid V := E_h[\cdot] \mid\ !E_h[\cdot] \mid$
$\quad \mathtt{raise}\, E_h[\cdot] \mid \mathtt{handle}\, E_h[\cdot]\, M \mid \mathtt{handle}\, k\, E_h[\cdot]\ (k \neq h)$
(Call-by-value) evaluation of programs is given by small-step reduction in an environment $\mathcal{E}$ consisting of a set of exception names, a set of reference names or locations $\mathcal{E}[L]$ and a state $\mathcal{E}[S]$, which is a partial map from L to values of λ_{RCE}.

Definition 2. *Operational semantics of λ_{RCE} is given by the reflexive transitive closure of the following relation between terms* of type $\mathbf{0}$ *in an environment $\mathcal{E}$ (containing a set of locations $\mathcal{E}[L]$, store $\mathcal{E}[S]$ and set of exception names $\mathcal{E}[Ex]$):*

$$E[\lambda x.M\ V], \mathcal{E} \longrightarrow E[M[V/x]], \mathcal{E}$$
$$E[\mathtt{new}], \mathcal{E}[L] \longrightarrow E[x], \mathcal{E}[L \cup \{x\}] : x \notin \mathcal{E}[L]$$
$$E[x := V], \mathcal{E}[S] \longrightarrow E[*], \mathcal{E}[S[x \mapsto V]] : x \in \mathcal{E}[L]$$
$$E[!x], \mathcal{E}[S] \longrightarrow E[S(x)], \mathcal{E}\ \ : S(x)\downarrow$$
$$E[\mathtt{new\_exn}], \mathcal{E}[Ex] \longrightarrow E[h], \mathcal{E}[Ex \cup \{h\}] :\ \ h \notin \mathcal{E}[Ex]$$
$$E[\mathtt{handle}\, h\, *], \mathcal{E} \longrightarrow E[*], \mathcal{E} : h \in \mathcal{E}[Ex]$$
$$E[\mathtt{handle}\, h\, E_h[\mathtt{raise}\, h]], \mathcal{E} \longrightarrow E[*], \mathcal{E} h \in \mathcal{E}[Ex]$$
$$E[\mathtt{callcc}\, V], \mathcal{E} \longrightarrow E[V\ \lambda x.\mathtt{abort}\, E[x]], \mathcal{E}$$
$$E[\mathtt{abort}\, M], \mathcal{E} \longrightarrow M, \mathcal{E}$$

For a program (closed term) $M : \mathtt{unit}$*, let* $\kappa : \mathtt{unit} \Rightarrow \mathbf{0}$ *be a variable not occurring free in* M*, then* $M \Downarrow$ *if* $\kappa\, M, \{\} \twoheadrightarrow \kappa *, \mathcal{E}$*.*

The standard notions of contextual approximation and equivalence will be used:
$M \lesssim_X N$ if for all closing λ_X-contexts, $C[M] \Downarrow$ implies $C[N] \Downarrow$.
$M \simeq_X N$ if $M \lesssim_X N$ and $N \lesssim_X M$.

Notation: $M; N$ will be used for $(\lambda x.N)\, M$ $(x \notin FV(N)$, $\mathtt{let}\, x = N\, \mathtt{in}\, M$ for $(\lambda x.M)\, N$, $\mathtt{new\_exn}\, h.M$ for $(\lambda x.M)\, \mathtt{new\_exn}$ and $\mathtt{new}\, x := V.M$ for $(\lambda x.x := V; M)\, \mathtt{new}$. At each type T there is a divergent term $\bot^T : T = \mathtt{new}_{\mathtt{unit} \Rightarrow T}\, y.y := (\lambda x.!y\, x).!y\, *$.

2.1 Macro-expressiveness: Some Simple Examples

The common basis of the various comparisons [11,7,1] of expressiveness is the notion of a *reduction* or translation between programming languages.

Definition 3. *A reduction from* $\mathcal{L}_1$ *to* $\mathcal{L}_2$ *is a (recursively definable) mapping* $\phi : \mathrm{Tm}_{\mathcal{L}_1} \to \mathrm{Tm}_{\mathcal{L}_2}$ *such that:*

- *if* $M \in \mathrm{Prog}_{\mathcal{L}_1}$ *then* $\phi(M) \in \mathrm{Prog}_{\mathcal{L}_2}$,
- $\phi(M) \Downarrow_{\mathcal{L}_2}$ *if and only if* $M \Downarrow_{\mathcal{L}_1}$.

ϕ *is* compositional *if it extends to a map on contexts such that* $\phi(C[M]) = \phi(C)[\phi(M)]$.

However, existence of such a reduction (whether compositional or not) merely amounts to the possibility of writing an interpreter for $\mathcal{L}_1$ in $\mathcal{L}_2$. As a test of expressiveness it is unlikely to be sufficient to distinguish between Turing-complete languages. A finer notion of relative expressiveness can be obtained by introducing additional criteria for determining a suitable notion of translation, such as the requirement that a reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$ should preserve their common structure in the following sense.

Definition 4. *Let* ϕ *be a reduction from* $\mathcal{L}_1$ *to* $\mathcal{L}_2$*, and* $\mathcal{L} \subseteq \mathcal{L}_1, \mathcal{L}_2$*. Then* ϕ preserves $\mathcal{L}$-contexts *if for all contexts* $C[\cdot]$ *of* $\mathcal{L}$*,* $\phi(C[M]) = C[\phi(M)]$*. If* $\mathcal{L} = \mathcal{L}_1 \cap \mathcal{L}_2$ *we shall just say that* ϕ preserves contexts.

The strength of this condition is clearly directly related to the content of $\mathcal{L}$; when the two languages are disjoint it has no force whereas when $\mathcal{L}_2 \subseteq \mathcal{L}_1$ it is equivalent to the notion of *eliminability* [1]; constructors $F_1, \ldots, F_n$ in a language $\mathcal{L}$ are *eliminable* if there is a translation ϕ from $\mathcal{L}$ to $\mathcal{L} - \{F_1, \ldots F_n\}$ such that for every $G \notin \{F_1, \ldots, F_n\}$, $\phi(G(M_1 \ldots M_n)) = G(\phi(M_1) \ldots \phi(M_n))$ — i.e. ϕ preserves $\mathcal{L}$ contexts.
$F_1, \ldots F_n$ are *macro-eliminable* if in addition each F_i is expressible as a "syntactic abstraction"; a context $A_i[\cdot] \ldots [\cdot]$ of $\mathcal{L} - \{F_1, \ldots, F_n\}$ such that $\phi(F_i(M_1 \ldots M_n)) = A_i[(\phi(M_1)] \ldots [\phi(M_n)]$ — i.e. ϕ is compositional. We shall use syntactic abstractions for defining compositional translations.

As an example we shall first show that the forms of `raise` and `handle` used here have the full expressive power of Gunther, Rémy and Riecke's simple exceptions [3] by giving syntactic abstractions for the latter. Simple exceptions differ from our "even simpler" exceptions in that they can carry values; there is a type-constructor $\_\,\texttt{exn}$ and at each type T there is an operation $\texttt{raise}_T$, typable as follows:

$$\frac{\Gamma \vdash M : T\,\texttt{exn} \quad \Gamma \vdash N : T}{\Gamma \vdash \texttt{raise}_T\, M\, N : S}$$

In the presence of state, exceptions carrying values of type T can be expressed by storing the latter in a reference of type $T\,\texttt{ref}$ and raising and handling an associated value of type `exn`.

The second difference between simple exceptions and those in λ_{RCE} is that the handle operation for the former applies a handler function when it catches an exception: the simple-exception handler $\texttt{handle}\, L\, \texttt{with}\, M\, \texttt{in}\, N$ has the typing rule:

$$\frac{\Gamma \vdash L : S\,\texttt{exn} \quad \Gamma \vdash M : S \Rightarrow T \;\; \Gamma \vdash N : T}{\Gamma \vdash \texttt{handle}\, L\, \texttt{with}\, M\, \texttt{in}\, N : T.}$$

The operational semantics for simple exceptions is given by an appropriate notion of evaluation context (see [3]), and the evaluation rules:

$$E[\texttt{handle}\, h\, \texttt{with}\, V\, \texttt{in}\, U], \mathcal{E} \longrightarrow E[U], \mathcal{E}$$
$$E[\texttt{handle}\, h\, \texttt{with}\, V\, \texttt{in}\, E_h[\texttt{raise}_T\, h\, U]], \mathcal{E} \longrightarrow E[V\, U], \mathcal{E}.$$

We can simulate simple exception handling by raising an additional exception which escapes from the handler function if the main body of the program evaluates without raising an exception.

Proposition 1. *Simple exceptions are macro-eliminable in* $\mathcal{L}$.

Proof. The following syntactic abstractions for simple exceptions simulate the reduction rules appropriately, and generate an equivalent notion of evaluation context:
$\phi(T\,\texttt{exn}) = (\phi(T)\,\texttt{ref} \Rightarrow (\texttt{exn} \Rightarrow \texttt{unit})) \Rightarrow \texttt{unit}$,
$\phi(\texttt{new\_exn}_T) = \texttt{new\_exn}\, x.\texttt{new}_T\, y.\lambda g.((g\, y)\, x)$,
$\phi(\texttt{raise}_T M\, N) = \phi(M)\, (\lambda xy.y := \phi(N); \texttt{raise}\, x)$,
$\phi(\texttt{handle}\, L\, \texttt{with}\, M\, \texttt{in}\, N) = \texttt{let}\, \phi(L) = l, \phi(M) = m, k = \texttt{new\_exn}, z = \texttt{new}_T \texttt{in}$
$(\texttt{handle}\, k\, (l\, \lambda xy.\texttt{handle}\, y\, (z := \phi(N); \texttt{raise}\, k); z := m\, !x)); !z$

3 Interference between Control Effects

Another sense in which a translation may preserve program structure is as follows.

Definition 5. *If* $\mathcal{L} \subseteq \mathcal{L}_1, \mathcal{L}_2$, *then* $\phi : \mathcal{L}_1 \to \mathcal{L}_2$ preserves $\mathcal{L}$-terms in context *if it extends to* $\mathcal{L}_1$*-contexts and for all contexts* $C[\cdot]$ *of* $\mathcal{L}_1$ *and all terms* M *of* $\mathcal{L}$, $\phi(C[M]) = \phi(C)[M]$.

Lemma 1. *If ϕ is compositional* and *preserves $\mathcal{L}$-contexts then ϕ preserves $\mathcal{L}$-terms in context.*

Proof. For any $\mathcal{L}$ term M, $\phi(M) = M$ (as M is a 0-ary $\mathcal{L}$-context) and hence $\phi(C[M]) = \phi(C)[\phi(M)] = \phi(C)[M]$.

A translation which preserves terms in context will also preserve observational equivalences — this is the basis for a useful test given in [1]; a necessary condition for a compositional and context-preserving reduction to exist.

Proposition 2. *If there is a reduction $\phi : \mathcal{L}_2 \to \mathcal{L}_1$ which preserves $\mathcal{L}$-terms in context then for all M_1, M_2 in $\mathcal{L}$, $M \lesssim_{\mathcal{L}_1} N$ implies $M \lesssim_{\mathcal{L}_2} N$.*

Proof. For any $\mathcal{L}_2$ context $C[\cdot]$, $C[M] \Downarrow$ implies $\phi(C[M]) = \phi(C)[M] \Downarrow$ implies $\phi(C)[N] = \phi(C[N]) \Downarrow$ implies $C[N] \Downarrow$.

Our first example showing that exceptions cannot be expressed using continuations and references is of this form; we shall show that exceptions can be used to break a simple and natural equivalence which holds in λ_{RC}. Moreover, an equivalence which is at the basis of several "equational theories of control", such as Felleisen's $\lambda\mathcal{C}$-calculus [2], and Parigot's $\lambda\mu$-calculus [9] (which has been proposed as a "a metalanguage for functional computation with control" by Ong and Stewart [8]).

Proposition 3. *For any $E[\cdot] : S$, $M : T$ in λ_{RC}:*
$E[\texttt{callcc}\, M] \simeq_{RC} \texttt{callcc}\, \lambda k^{S \Rightarrow T}.E[M\ \lambda y.k\ E[y]]$

This equivalence is a typed version of the rule C_{lift} which is a key axiom of Sabry and Felleisen's equational theory of the λ-calculus with `callcc` [13], where it is shown to be sound using a cps translation. To prove that it holds in λ_{RC}, we use an approximation relation. Let $\sim$ be the least *congruence* on terms of λ_{RC} such that for all evaluation contexts $E[\cdot] : S$ and $M : T$, $E[\texttt{callcc}\, M] \sim \texttt{callcc}\, \lambda k^{S \Rightarrow T}.E[M\ \lambda y.k\ E[y]]$,
and for all $E[\cdot]$ such that if x is not free in $E[\cdot]$ or M, $E[M] \sim \lambda x.E[x]\, M$.
We extend $\sim$ straightforwardly to a relationship on environments: $\mathcal{E} \sim \mathcal{E}'$ if $\mathcal{E}[L] = \mathcal{E}'[L]$ and for all $x \in \mathcal{E}[L]$, $\mathcal{E}[S](x) \sim \mathcal{E}'[S](x)$.

Lemma 2. *If $M, \mathcal{E} \sim M', \mathcal{E}'$ and $M', \mathcal{E}' \longrightarrow M'', \mathcal{E}''$ then $\exists \widehat{M}, \widehat{\mathcal{E}}$ such that $M, \mathcal{E} \twoheadrightarrow \widehat{M}, \widehat{\mathcal{E}}$ and $\widehat{M}, \widehat{\mathcal{E}} \sim M'', \mathcal{E}''$.*

Corollary 1. *If $M \sim M'$, then $M \Downarrow$ if and only if $M' \Downarrow$.*

To prove Proposition 3, it suffices to observe that for any $C[\cdot]$, $C[E[\texttt{callcc}\, M]] \sim C[\texttt{callcc}\, \lambda k^{S \Rightarrow T}.E[M\ \lambda y.k\ E[y]]]$ and so by Lemma 2, $C[E[\texttt{callcc}\, M]] \Downarrow$ if and only if $C[\texttt{callcc}\, \lambda k^{S \Rightarrow T}.E[M\ \lambda y.k\ E[y]]] \Downarrow$.

However, because exceptions and continuations both manipulate the flow of control they can "interfere" with each other, breaking this equivalence, *even between terms which do not contain exceptions.* For instance, suppose f is a variable of type $(\texttt{unit} \Rightarrow \texttt{unit}) \Rightarrow \texttt{unit} \Rightarrow (\texttt{unit} \Rightarrow \texttt{unit})$. Then we have $(\texttt{callcc}\, f) * \simeq_{RC} \texttt{callcc}\, \lambda k.(f\ \lambda y.k\ (y\ *))\ *$, as this is an instance of the equivalence proved in Proposition 3, with $E[\cdot] = [\cdot]\ *$.

Proposition 4. $\lambda f.((\texttt{callcc}\, f)\, *) \not\simeq_{RCE} \lambda f.\texttt{callcc}\, \lambda k.((f\, \lambda y.k\, (y\, *))\, *)$.

Proof. Let $N \equiv \lambda gx.(\texttt{handle}\, h\, (g\, \lambda v.\texttt{raise}\, h); \texttt{raise}\, e$.
Then $(\texttt{callcc}\, N)\, *$ raises exception h but $\texttt{callcc}\, \lambda k.(N\, \lambda y.k\, (y\, *))\, *$ raises exception e, and so if:
$C_1[\cdot] \equiv \texttt{new\_exn}\, h, e.\texttt{handle}\, h\, (\lambda f.[\cdot]\, N)$,
$C_2[\cdot] \equiv \texttt{new\_exn}\, h, e.\texttt{handle}\, e\, (\lambda f.[\cdot]\, N)$,
then $C_1[E[\texttt{callcc}\, f]] \Downarrow$ and $C_1[\texttt{callcc}\, \lambda k.E[f\, \lambda y.k\, E[y]]] \not\Downarrow$,
but $C_2[\texttt{callcc}\, \lambda k.E[f\, \lambda y.k\, E[y]]] \Downarrow$ and $C_2[E[\texttt{callcc}\, f]] \not\Downarrow$.

Corollary 2. *Exceptions are not macro-eliminable in* λ_{RCE}.

The fact that exceptions cannot be expressed in control calculi based on first-class continuations such as $\lambda\mathcal{C}$ or $\lambda\mu$ has already been shown in [12]. But the result given here is stronger — these calculi are not even sound for reasoning about exception-free programs if there is the possibility that they might interact with exceptions. This is an important point of difference between control calculi and the (call-by-value) λ-calculus, which is notable for its robustness in the presence of side-effects.

4 Implementing Exceptions with Continuations

We have established that exceptions, continuations and references cannot be satisfactorily reduced to continuations and references, but this leaves open the problem of whether exceptions and references can be reduced to exceptions and continuations. In other words, is there a translation from λ_{RE} into λ_{RC} which preserves only the terms or contexts of λ_R? The existence of even a limited reduction of this kind would lend some plausibility to the claim that continuations (with references) are more expressive than exceptions, because it is known not to be possible to reduce continuations and references to exceptions and references [14].

Moreover, it is possible to give alternative operational semantics of exceptions combined with continuations. For example, New Jersey SML includes an additional type constructor — `control_cont`— for "primitive continuations" which ignore enclosing exception handlers, and control operators — `capture` and `escape` — corresponding to `callcc` and `throw`, for manipulating them. For programs without exceptions, substituting `capture` and `escape` for `callcc` and `throw` yields an equivalent program, but this is not true in the presence of exceptions, and in fact the counterexample of Section 3 is not valid for primitive continuations.

However, exceptions *cannot* break any equivalence between λ_R terms which is not broken by continuations and references, because there is an implementation of exceptions using continuations and references which preserves terms of λ_R in context. This implementation is essentially as described in [10]. Exception names are represented as references of type $(\texttt{unit} \Rightarrow \mathbf{0}) \Rightarrow (\texttt{unit} \Rightarrow \mathbf{0}) \Rightarrow \mathbf{0}$ — they are not used to store anything, but can be tested for equality — define:

If $M = N$ then L else $L' \equiv (M := \lambda xy.y\,*); (N := \lambda xy.x\,*); (!M\ \lambda z.L\ \lambda z.L')$.
The current continuation of each handler is stored in a stack, which is represented as "handler function" inside a reference variable exh. Raising an exception simply applies the value of exh to the exception name, which then replaces the current continuation with the relevant handler continuation, and resets exh. Non-compositionality of the implementation stems from the global nature of exh; access to this variable must be shared by all parts of a term, but it must be initialized at the start of each program.
Thus the implementation can be represented as a translation ψ on *terms* of λ_{RE} defined by the following syntactic abstractions:
$\psi(\texttt{new\_exn}) = \texttt{new}$,
$\psi(\texttt{handle}\ M\ N) = \texttt{let}\ old\ = !exh$
$\texttt{in callcc}\ \lambda k.exh := \lambda y.\text{If}\ \ \psi(M) = y\ \ \text{then}\ (exh := old; (k\ \psi(N)))$
$\text{else}\ (old\ y))$,
$\psi(\texttt{raise}\ M)) = !exh\ \psi(M)$.
This yields a translation ϕ on *programs*: $\phi(M) = \texttt{new}\, exh := \lambda x.\bot.\psi(M)$ such that if $\phi(C)[\cdot] =_{df} \texttt{new}\, exh := \lambda x.\bot.\psi(C)[\cdot]$ then ϕ preserves λ_R terms-in-context.

Proposition 5. *For any program M of λ_{RE}, $M \Downarrow$ if and only if $\phi(M) \Downarrow$.*

Proof. Define ψ as a map on evaluation contexts as follows:
$\psi([\cdot]) = [\cdot], \psi(E[V\ [\cdot]]) = \psi(E)[\psi(V)\ [\cdot]], \ldots,$
$\psi(E[\texttt{raise}\ [\cdot]]) = \psi(E)[(!exh\ [\cdot]], \psi(E[\texttt{handle}\ h\ [\cdot]]) = \psi(E)[[\cdot]; !exh\ h]$.
This map is used to define an operation $\ulcorner E[\cdot] \urcorner$ which extracts the current contents of exh, represented as a list of pairs $(h, E[\cdot])$ of names and handler contexts:
$\ulcorner [\cdot] \urcorner = [\,], \ulcorner E[V\ [\cdot]] \urcorner = \ulcorner E[\cdot] \urcorner, \ldots, \ulcorner E[\texttt{handle}\ h\ [\cdot]] \urcorner = \ulcorner E[\cdot] \urcorner :: (h, \psi(E[\cdot]))$.
An inductive proof of soundness can then be based on the following facts:
$E[\psi(E'[M])], \mathcal{E}[exh \mapsto l] \twoheadrightarrow E[\psi(E')[\psi(M)], \mathcal{E}[exh \mapsto l :: \ulcorner E'[\cdot] \urcorner]$,
and if $E[M], \mathcal{E} \longrightarrow E[M'], \mathcal{E}'$ then:
$\psi(E)[\psi(M)], \mathcal{E}[exh \mapsto l] \twoheadrightarrow \psi(E)[\psi(M')], \mathcal{E}'[exh \mapsto l]$.

Corollary 3. *Equivalence of λ_R terms with respect to λ_{RCE} contexts is conservative over equivalence of λ_R terms with respect to λ_{RC} contexts.*

Thus the implementation cannot be soundly extended to one which preserves λ_{RC} terms in context; the proof of Proposition 4 provides a counterexample — $\phi(C_2)[(\texttt{callcc}\ f)\ *]$ converges.

5 Expressiveness and First-Order Formulas

Does Corollary 3 entail that exceptions can be expressed using continuations and references if we don't have continuations in our source language? We might reasonably take the fact that the implementation of exceptions preserves λ_R equivalences to be *the* sufficient condition for it to be a satisfactory reduction of λ_{RE} to λ_{RC}. However, we shall show that no translation from λ_{RE} to λ_{RC} can

exist which adheres to our original criteria — compositionality and preservation of contexts — as such a translation will preserve the truth of all Π_1 statements which do not mention exceptions, whereas there is a such a statement which is true in λ_{RC} but not in λ_{RE}.

Definition 6. *For a programming language $\mathcal{L}$, let the* object language of $\mathcal{L}$, $\mathrm{obj}(\mathcal{L})$*, be the language of first-order logic with two unary predicates* Prog *and* Eval *and terms generated from $\Sigma_{\mathcal{L}}$, together with a distinct set of* logical *variables* $\mathsf{x}, \mathsf{y}, \mathsf{z}, \ldots$.
Let $\mathcal{M}(\mathcal{L})$ be the $\mathrm{obj}(\mathcal{L})$*-structure with the domain* $\mathrm{Tm}_{\mathcal{L}}$ *in which each term of* $\mathrm{obj}(\mathcal{L})$ *is interpreted as the corresponding term of $\mathcal{L}$, and $\mathcal{M}(\mathcal{L}) \models \mathsf{Prog}(t)$ if and only if $t \in \mathrm{Prog}_{\mathcal{L}}$ and $\mathcal{M}(\mathcal{L}) \models \mathsf{Eval}(t)$ if and only if $t \in \mathrm{Eval}_{\mathcal{L}}$.*

Proposition 6. *If there is a compositional and context-preserving translation $\phi : \mathcal{L}_1 \to \mathcal{L}_2$ then for all Π_1 sentences of* $\mathrm{obj}(\mathcal{L}_1 \cap \mathcal{L}_2)$*, if $\mathcal{M}(\mathcal{L}_2) \models \theta$ then $\mathcal{M}(\mathcal{L}_1) \models \theta$.*

Proof. If $\mathcal{M}(\mathcal{L}_1) \not\models \forall y_1 \ldots y_n.\theta(y_1, \ldots, y_n)$ then there exist terms $M_1, \ldots M_n$ in $\mathcal{L}_1$ such that $\mathcal{M}(\mathcal{L}_1) \models \neg\theta(M_1, \ldots, M_n)$. It is then straightforward to show by structural induction that $\mathcal{M}(\mathcal{L}_2) \models \neg\theta(\phi(M_1), \ldots, \phi(M_n))$, and hence $\mathcal{M}(\mathcal{L}_2) \not\models \forall y_1 \ldots y_n.\theta$ as required.

So to show that there is no compositional and context-preserving reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$, it is sufficient to find a Π_1 sentence θ of $\mathrm{obj}(\mathcal{L}_1 \cap \mathcal{L}_2)$ such that $\mathcal{M}(\mathcal{L}_2) \models \theta$ and $\mathcal{M}(\mathcal{L}_1) \not\models \theta$. In a λ-calculus-based language this *includes* all counterexamples in the form of a contextual equivalence of $\mathcal{L}_2$ which is broken in $\mathcal{L}_1$ since contextual equivalence of values U, V can be expressed in $\mathrm{obj}(\mathcal{L})$ by the Π_1 sentence $\forall \mathsf{x}.\mathsf{Eval}(\mathsf{x}\,U) \iff \mathsf{Eval}(\mathsf{x}\,V)$. But we gain access to new counterexamples which are not of this form; we shall give a context $C[\cdot]$ and a value U of λ_R such that for $\theta \equiv \forall \mathsf{x}, \mathsf{y}.\mathsf{Prog}(C[\mathsf{x}]) \wedge \mathsf{Eval}(\mathsf{y}\,C[\mathsf{x}]) \Longrightarrow \mathsf{Eval}(\mathsf{y}\,U)$ we have $\mathcal{M}(\lambda_{RC}) \models \theta$ and $\mathcal{M}(\lambda_{RE}) \not\models \theta$.

Let $T = (\mathtt{unit} \Rightarrow \mathbf{0}) \Rightarrow \mathbf{0}$, $U = \lambda g : T.g$, and $C[\cdot] \equiv V\,\lambda f : T.([\cdot] : \mathbf{0})$, where $V \equiv \lambda F.\lambda g.\lambda x.\mathtt{new}\, z := \lambda a.((z := \lambda y.x\,*); g\,a).F\,\lambda w.!z\,w$. So θ represents the assertion that for any $M : \mathbf{0}$ containing only $f : (\mathtt{unit} \Rightarrow \mathbf{0}) \Rightarrow \mathbf{0}$ free, $V\,\lambda f.M \lesssim \lambda g.g$.

Proposition 7. $\mathcal{M}(\lambda_{RE}) \not\models \theta$.

Proof. Let $M \equiv \mathtt{new\_exn}\, h.(\mathtt{handle}\, h\, (f\,\lambda x.\mathtt{raise}\, h); \bot); f\,\lambda w.\bot$ and $D[\cdot] \equiv \mathtt{new\_exn}\, k.\mathtt{handle}\, k\, ((([\cdot]\,\lambda x.\mathtt{handle}\, k\, (x\,*); \bot)\,\lambda y.\mathtt{raise}\, k); \bot)$.
Then $D[V\,\lambda f.M] \Downarrow$ but $D[\lambda g.g] \not\Downarrow$.

The proof that $\mathcal{M}(\lambda_{RC}) \models \theta$ can be outlined via a series of lemmas about $\lesssim_{RC}$ and $\simeq_{RC}$. First we show that terms of type $\mathbf{0}$ containing only $f : T \Rightarrow \mathbf{0}$ free can be reduced to a "head-normal form".
Write $\mathtt{new}\,\overline{y} := \overline{v}.M$ for $\mathtt{new}\, y_1.\mathtt{new} y_2 \ldots \mathtt{new}\, y_n.y_1 := v_1; y_2 := v_2; \ldots; y_n := v_n.M$.

Lemma 3. *If $M : \mathbf{0}$ is a λ_{RC} term containing only $f : T \Rightarrow \mathbf{0}$ free, then there are some λ_{RC}-values $U : T, \overline{v}$ such that $M \lesssim_{RC} \mathtt{new}\,\overline{y} := \overline{v}.f\,U$.*

Proof. The following facts (proved using approximation relations) are used to inductively *reduce* terms of type $M : \mathbf{0}$ to head-normal form:
$E[\mathtt{abort}\,M] : \mathbf{0} \simeq_{RC} M$
$E[\mathtt{callcc}\,M] : \mathbf{0} \simeq_{RC} E[M\,\lambda x.\mathtt{abort}\,E[x]]$
$E[\mathtt{new}\,z.M] \simeq_{RC} \mathtt{new}\,z.E[M]$
$\mathtt{new}\,\overline{y} := \overline{v}.E[!y_i] \simeq_{RC} \mathtt{new}\,\overline{y} := \overline{v}.E[v_i]$
$\mathtt{new}\,\overline{y} := \overline{v}.E[y_i := U] \simeq_{RC}$
$\mathtt{new}\,y_1 := v_1 \ldots y_{i-1} := v_{i-1}.y_{i+1} := v_{i+1} \ldots y_n := v_n.y_i := U.E[*]$.
If this reduction process *does not* terminate, then M is equivalent to $\bot_{\mathbf{0}}$.

The following lemma is proved using approximation relations.

Lemma 4. **i** *If a is not free in W then:*
$\mathtt{new}\,z := \lambda a.((z := W); M).\mathtt{new}\,\overline{y} := \overline{v}.!z\,U$
$\simeq_{RC} \mathtt{new}\,z := W.\mathtt{new}\,\overline{y} := \overline{v}.M[U/a]$.
ii *For any term $M(f)$ and value V which do not contain z free:*
$\mathtt{new}\,z := V.M[\lambda y.!z\,y/f] \simeq_{RC} M[V/f]$.

The final lemma is a refinement of Lemma 3.

Lemma 5. *For any term $M \equiv \mathtt{new}\,\overline{y} := \overline{v}.\lambda w.N : \mathtt{unit} \Rightarrow \mathbf{0}$, containing only $x : \mathtt{unit} \Rightarrow \mathbf{0}$ free, $M \lesssim_{RC} x$.*

Proposition 8. *For any λ_{RC}-term $M(f) : \mathbf{0}$ which contains only $f : (\mathtt{unit} \Rightarrow \mathbf{0}) \Rightarrow \mathbf{0}$ free, $V\,\lambda f.M \lesssim_{RC} \lambda g.g$.*

Proof. By Lemma 3, there exist $U, \overline{v}$ such that $M \lesssim_{RC} \mathtt{new}\,\overline{y} := \overline{v}.f\,U$ and hence $V\,\lambda f.M$
$\lesssim_{RC} \lambda gx.\mathtt{new}\,z := \lambda a.(z := \lambda b.x\,*); g\,a.(\mathtt{new}\,\overline{y} := \overline{v}.f\,U)[\lambda w.!z\,w/f]$
$\simeq_{RC} \lambda gx.\mathtt{new}\,z := \lambda a.(z := \lambda b.x\,*); g\,a.(\mathtt{new}\,\overline{y} := \overline{v}.!z\,U)[\lambda w.!z\,w/f]$
$\lesssim_{RC} \lambda gx.\mathtt{new}\,z := \lambda b.x\,*.\mathtt{new}\,\overline{y} := \overline{v}.g\,U[\lambda w.!z\,w/f]$ (By Lemma 4.i)
$\simeq_{RC} \lambda gx.\mathtt{new}\,\overline{y} := \overline{v}.g\,U[\lambda b.x\,*/f]$ (By Lemma 4.ii)
$\simeq_{RC} \lambda gx.g\,(\mathtt{new}\,\overline{y} := \overline{v}.U)[\lambda b.x\,*/f]$
$\lesssim_{RC} \lambda gx.g\,x$ (By Lemma 5) $\simeq_{RC} \lambda g.g$.

Corollary 4. *There is no compositional and context-preserving reduction from λ_{RE} to λ_{RC}.*

6 Conclusions

What relevance do these results have to the design, implementation and application of programming languages? Whilst expressiveness can mean the facility to write concise, flexible and efficient programs, the kind of expressive power which is embodied in our counterexamples does not appear to be particularly useful. Indeed, quite the reverse — combining continuations and exceptions gives the

"power" to write programs with unpredictable behaviour, and this should be balanced against the usefulness of these effects when permitting such combinations. A better way to combine the simplicity of exceptions with the power of continuations could be to provide dynamically bound control constructs which still allow complex, continuation-style behaviour, such as *prompts* [3].

The difficulty of predicting on an ad hoc basis how control effects will interact suggests that more formal ways of reasoning about them would be useful. One possibility is equational reasoning using "control calculi" such as $\lambda\mathcal{C}$ [2] or $\lambda\mu$ [8]. The counterexample in Section 3 shows the limitations of these calculi, however, in that their equational theories are not consistent with the presence of exceptions.

There are many other ways to model or reason about control, but one which deserves mention is *game semantics*. The results described here arose from a semantic study of exceptions and continuations in a fully abstract games model [4,5]. Thus one of the conclusions they support is a methodological one; game semantics — with its focus on definability and full abstraction — can be a useful tool for investigating relative expressiveness. Moreover game-based reasoning can be readily converted into syntactic examples (using definability results) which can be understood in isolation from the semantics.

Acknowledgments

I would like to thank Guy McCusker and the referees for their comments.

References

1. Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, volume 17, pages 35–75, 1991. 133, 133, 134, 135, 137, 137, 139
2. Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205 – 207, 1987. 134, 139, 144
3. C. Gunter, D. Rémy, and J. Riecke. A generalization of exceptions and control in ML like languages. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pages 12–23, 1995. 133, 136, 138, 138, 144
4. J. Laird. Full abstraction for functional languages with control. In *Proceedings of the Twelfth International Symposium on Logic In Computer Science, LICS '97*. IEEE Computer Society Press, 1997. 144
5. J. Laird. A fully abstract game semantics of local exceptions. In *Proceedings of the Sixteenth International Symposium on Logic In Computer Science, LICS '01*. IEEE Computer Society Press, 2001. 144
6. M. Lillibridge. Unchecked exceptions can be strictly more powerful than Call/CC. *Higher-Order and Symbolic Computation*, 12(1):75–104, 1999. 133
7. J. Mitchell. On abstraction and the expressive power of programming languages. In *Proc. Theor. Aspects of Computer Software*, pages 290–310, 1991. 133, 137

8. C.-H. L. Ong and C. Stewart. A Curry-Howard foundation for functional computation with control. In *Proceedings of ACM SIGPLAN-SIGACT syposium on Principles of Programming Languages, Paris, January 1997*. ACM press, 1997. 139, 144
9. M. Parigot. $\lambda\mu$ calculus: an algorithmic interpretation of classical natural deduction. In *Proc. International Conference on Logic Programming and Automated Reasoning*, pages 190–201. Springer, 1992. 134, 139
10. J. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998. 134, 140
11. Jon G. Riecke. *The Logic and Expressibility of Simply-Typed Call-by-Value and Lazy Languages*. PhD thesis, Massachusetts Institute of Technology, 1991. Available as technical report MIT/LCS/TR-523 (MIT Laboratory for Computer Science). 133, 137
12. J. Riecke and H. Thielecke. Typed exceptions and continuations cannot macro-express each other. In J. Wiedermann, P. van Emde Boas and M. Nielsen, editor, *Proceedings of ICALP '99*, volume 1644 of *LNCS*, pages 635 –644. Springer, 1999. 134, 134, 134, 140
13. A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3/4):289–360, 1993. 139
14. H. Thielecke. On exceptions versus continuations in the presence of state. In *Proceedings of ESOP 2000*, volume 1782 of *LNCS*. Springer, 2000. 134, 134, 134, 134, 140

Appendix

```
(* SML_NJ code corresponding to sections 3 - 5.*)
(* Counterexample from section 3: *)
open SMLofNJ.Cont;
exception E;exception F
fun M y x = ((y (fn z => raise E)) handle E =>();raise F);
(* (callcc (fn k => (M (fn x => throw k x)))) ();
   raises exception E whereas
   callcc (fn k =>  ((M (fn x => throw k (x ())))  ()));
   raises exception F *)

(* Implementation of exceptions  described in section 4:*)
fun diverge x = diverge x;
val exhandler = ref (fn x:unit ref => (diverge ()):unit);
val new_exn M = = M (ref ());
fun handle_xn h x = let val old = !exhandler in
                    (callcc (fn k =>
                    ((exhandler := (fn y =>
                    (if (y = h)
                       then ((exhandler:= old);(throw k ()))
                       else  (old y))));
                    ((fn v => ((exhandler:= old);v)) (x ()))))))
 end;
```

```
(*It's necessary to ``thunk'' the second argument to handle_xn.*)
fun raise_xn h  = diverge (!exhandler h);

(*Counterexample from section 5: *)
datatype Empty = empty of Empty;
fun V g (f: (unit -> Empty) -> Empty) (x:unit -> Empty) =
let val z = (ref diverge) in
  ((z:= (fn u => ((z:= (fn y => x ()));(f u))));
  ((diverge (g (fn w => (!z w))))):Empty
 end;
fun N f = let exception H in
          (diverge (f (fn w=>raise H)) handle H=>();f diverge)
end;
fun arg1 x = diverge (diverge (x ()) handle F => ());
fun arg2 z = raise F;
(*diverge (((V N) arg1) arg2) handle F=>(); converges, *)
(*diverge (((fn g => g) arg1) arg2) handle F =>(); diverges.*)
```