Contrasting Exceptions and Continuations

Hayo Thielecke
H.Thielecke@cs.bham.ac.uk
School of Computer Science
University of Birmingham
Birmingham B15 2TT
United Kingdom

October 19, 2001

Abstract

Exceptions and first-class continuations are the most powerful forms of control in programming languages. While both are a form of non-local jumping, there is a fundamental difference between them. We formalize this difference by proving a contextual equivalence that holds in the presence of exceptions, but not continuations; and conversely, we prove an equivalence that holds in the presence of continuations, but not exceptions. By the same technique, we show that exceptions and continuations together do not give rise to state, and that exceptions and state do not give rise to continuations.

1 Introduction

Practically all programming languages contain constructs for manipulating the control flow by jumping. The most powerful of these control constructs are exceptions and continuations. Exceptions are very widespread and are part of many modern languages, like ML [15, 13], Java [6] and C++ [23]. Continuations in the form of the `call/cc`-operator are part of the standard of only a single language, Scheme [9], and they also appear in the New Jersey dialect of Standard ML. Despite their relative scarcity as a programming language feature, however, continuations are of interest as one of the fundamental notions of programming languages [21, 22].

Control operators comparable to exception arose in Mac Lisp in the form of the `catch` and `throw` operations [20]. As early Lisp used dynamic scope, their semantics was naturally dynamic too. (Such an archaic, entirely dynamic, Lisp may now be most accessible to most readers in the form of Emacs Lisp.) When dynamic scope was replaced by static scope in Common Lisp, the dynamic semantics of `catch`/`throw` was retained. Similarly, ML, while otherwise strongly based on static scoping and typing, has dynamic control in the form of exceptions. The reason for this dynamic semantics is pragmatic: the main application of dynamic control is error handling, and errors are best handled dynamically when they occur.

Continuations, by contrast, are from the outset strongly associated with static binding, arising as the denotations of labels. Control operators for first-class continuations are even older than the abstract concept of continuation. When Landin explicated the semantics of procedures by way of closures in the SECD machine, it was natural to introduce control by way of a special form of closure, giving a statically scoped form of control, namely Landin's **J**-operator [11].

In some cases, exceptions and continuations can be used more or less interchangeably. For example, textbooks (such as Felleisen and Friedman's "Seasoned Schemer" [5]) usually introduce continuations with simple idioms that could equally well be programmed with exceptions (see [24] for a discussion). This should not be taken to mean that exceptions and continuations are the same: in more advanced examples the difference between them manifest themselves.

There are some subtle pitfalls in understanding exceptions and continuations, which can lead one to confuse them. In fact, examples of such confusion, namely wrong encodings of exceptions in terms of continuations, have appeared in the literature. Concretely, if one is not already familiar with both exceptions and continuations, then one or both of the following two assertions may sound reasonable, even obvious:

- For exceptions: consider M `handle e =>` N in ML, or `(catch 'e` M`)` in Lisp. Then `e` is bound in M.
- For continuations: consider `callcc` $(\lambda k.M)$. Then the continuation bound by the `callcc`$(\lambda k. \ldots)$ can never escape from the scope of the λ.

As the astute reader may have noticed, both assertions are in fact glaringly false, mainly due to a confusion of scope and extent. But interestingly, they are false in a complementary fashion that illustrates the difference between the constructs. Confusing handlers with binders ascribes to exceptions a continuation-like semantics; whereas limiting the extent of the continuation to the scope of the variable k would confine continuations to a more exception-like behaviour.

More generally, one of the central points of this paper is that the contrast between exceptions and continuations elucidates both constructs. The dynamic nature of exceptions becomes clearer when contrasted with the static one of continuations. Conversely, the more advanced usages of upward continuations and backtracking are those that are beyond the capabilities of exceptions.

Our criterion for the difference between exceptions and continuations can be explained quite intuitively. It was in fact already mentioned informally by Landin [11, 12] in the same paper that introduced the **J**-operator, a forebear of `call/cc`. Landin had eliminated many programming language features as inessential "syntactic sugar". For example, local variable binding, as with `let` in Scheme or ML, is syntactic sugar for λ-abstraction followed by application. Some constructs however are not eliminable this way: state and control. For state, Landin argued that assignment breaks the equivalence

$$\texttt{(cons (car } M\texttt{) (cdr } M\texttt{))} \cong M$$

In a purely functional language without state, the evaluation of M always produces the same result. If state is added, the equivalence can be broken. Therefore, the addition of state makes the language strictly more powerful. This notion of "expressive power" was later formalized by Felleisen [4]. At the very least, the increase in expressive power shows that one cannot write assignment as a macro; but it could also be argued that it does more than that, as it elucidates how assignment alters the character of the language and how one can reason about it. Another parallel to this early work by Landin is that the equivalences we use in the stateless setting are of the same flavour as the above one, in that they depend on whether evaluating the same expression once or twice makes a difference.

The main results can be summarized in terms of equivalences as follows:

Contextual equivalence			holds in the presence of
$(\lambda f.(f\,T; f\,F))M$	$\cong$	$(\lambda fg.(f\,T;\, g\,F))MM$	exceptions but not continuations
$(\lambda f.M)(\lambda x.O)$	$\cong$	$(\lambda f.M)(\lambda f.((\lambda f.M)(\lambda x.O); O))$	continuations but not exceptions
M	$\cong$	$(M; M)$	exceptions and continuations

Here T, F and O are meant to be arbitrary values. In particular, T and F can be values that we want to regard as distinct, such as $T = 1$ and $F = 0$. In a pure λ-calculus setting, one could also choose $T = \lambda xy.x$ and $F = \lambda xy.y$.

In an earlier (conference) version of some of this work, a big-step semantics was used to show the equivalence for exceptions, while a CPS transform was used for continuations [17]. The latter technique restricted the generality of the result. Here we try to use the most direct technique to prove the contextual equivalence. A small-step operational semantics is used for all constructs, and a simulation relation is constructed. Our interest here is not in proof technique for operational semantics as such, but rather in the impact of control (exceptions, continuations, or both) on what kind of reasoning is possible.

The `callcc`-operator gives the programmer direct access to what in semantics or implementation terms is called a continuation. We will abuse terminology somewhat by using the word "continuation" mainly in this sense, that is, first-class control objects introduced by calls of `callcc`. When we say that first-class continuations cannot express exceptions (in the absence of state), this is meant in the sense that `callcc` cannot express them. That should not be taken to imply that there can be no continuation semantics of exceptions; rather, it merely states that such a semantics does not factor over that of `callcc`. In fact, a continuation semantics for exceptions can easily be defined by passing two continuations, the current one and one for the current exception handler [2, 27]

1.1 Related work

While the literature on continuations is large, the semantics of exceptions has received less attention. An early example of a comparison of continuations and exceptions is Lillibridge [14]. Contextual equivalence as a tool for comparing programming languages goes back at least to Landin [11], and was later formalized by Felleisen [4]. Sitaram and Felleisen [18] used a contextual equivalence for showing the added power of control delimiters over `call/cc`. The continuations folklore knows an implementation of exceptions in terms of continuations and references (see [16]); hence there can be no equivalence broken by the former but not the latter, although the folklore implementation fails to be a macro-encoding [10].

The present paper is part of a broader study of control operators using category theory [24], classical logic [27] and linear typing [2]; it largely supersedes the earlier papers based on equivalences [17, 25, 26]. We have strengthened the results from [17], generalising to an untyped setting, removing a restriction to closed expressions, and making the proofs more uniform by using small-step semantics throughout.

1.2 Organization of the paper

To set the stage, we recall some of the basics of exceptions and continuations in Section 2. We then consider exceptions and continuations if each is added to a base language without state: exceptions cannot express continuations (Section 3); conversely, continuations in the absence of state cannot express exceptions (Section 4). If exceptions and continuations are combined in the same language, they cannot express state (Section 5). Exceptions and state cannot express continuations (Section 6) Section 7 concludes. Some technical lemmas are included as an appendix.

2 Exceptions and first-class continuations

We will need different programming language fragments with exceptions, continuations and state in various combinations. The syntax we use most closely resembles Standard ML of New Jersey, as this is the main language having both exceptions and continuations, as well as state. The language is equipped with an operational semantics that uses evaluation contexts to represent control, a style of semantics that was developed by Friedman, Felleisen and others, and which has become quite standard for reasoning with continuations. The reader familiar with such semantics may wish to skim this section.

We adopt some notational conventions: uppercase letters M, N, P, Q, ... range over expressions, lowercase letters x, y, ... range over variables. The substitution of x in M by N is written as $M[x \mapsto N]$. As is usual, we assume bound and free variables to be distinct, so that substitution does not lead to name-capture.

The dichotomy between exceptions and continuations becomes apparent if control is combined with functional features, so that the minimum setting we need for our results is λ-calculus. The functional core of the language is essentially call-by-value λ-calculus, with the following grammar:

$$M \quad ::= \quad x \mid MM \mid \lambda x.M$$

Because we assume call-by-value semantics, sequencing becomes syntactic sugar in terms of λ-abstraction. We write "$M;N$" for an expression that evaluates M, discards the result, and then proceeds with N, and define it as follows:

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

To make the language a little more concrete , we add also some basic arithmetic, conditional, and recursion, extending the grammar thus:

$$\begin{array}{rcl} M & ::= & \ldots \mid n \mid \texttt{succ}\,M \mid \texttt{pred}\,M \mid \texttt{if0}\,M\,\texttt{then}\,N\,\texttt{else}\,L \\ & \mid & \texttt{rec}\,f(x).\,M \end{array}$$

We then add control to the functional core. For continuations, the operations are `callcc` and `throw`, as in Standard ML of New Jersey. The grammar is extended as follows:

$$M \quad ::= \quad \ldots \mid \texttt{callcc}\,M \mid \texttt{throw}\,M\ N$$

The operator `callcc` seizes the current continuation, makes it into a first-class value, and passes this "reified" continuation to its argument. The argument being typically a λ-abstraction, one can read the idiom

$$\texttt{callcc}\,(\lambda k.M)$$

as "bind the current continuation to k in M". The operator `throw` invokes a continuation with an argument; `throw` k 42 is a jump to k, much like a `goto` in more traditional languages. In Scheme, the operation `call/cc` not only reifies the current continuation, but also wraps it into a procedure, so that it can be invoked by ordinary application. It is easy to translate from the notation used here to Scheme: one simply omits `throw`.

Definition 2.1 (λ_V+cont) Let λ_V+**cont** be the language defined by the operational semantics in Figure 1.

Figure 1: Operational semantics of λ_V+**cont**

$$\begin{array}{lcl}
V & ::= & x \mid n \mid \lambda x.M \mid \#E \\
E & ::= & [\,] \mid (E\ M) \mid (V\ E) \mid (\texttt{succ}\ E) \mid (\texttt{pred}\ E) \mid (\texttt{if0}\ E\ \texttt{then}\ M\ \texttt{else}\ M) \\
 & \mid & (\texttt{throw}\ E\ M) \mid (\texttt{throw}\ V\ E)
\end{array}$$

$$\begin{array}{lcl}
E[(\lambda x.\ P)\ V] & \to & E[P[x \mapsto V]] \\
E[\texttt{succ}\ n] & \to & E[n+1] \\
E[\texttt{pred}\ 0] & \to & E[0] \\
E[\texttt{pred}\ (n+1)] & \to & E[n] \\
E[\texttt{if0}\ 0\ \texttt{then}\ M\ \texttt{else}\ N] & \to & E[M] \\
E[\texttt{if0}\ (n+1)\ \texttt{then}\ M\ \texttt{else}\ N] & \to & E[N] \\
E[\texttt{rec}\ f(x).\ M] & \to & E[\lambda x.M[f \mapsto \texttt{rec}\ f(x).\ M]] \\
E[\texttt{callcc}\ M] & \to & E[M(\#E)] \\
E[\texttt{throw}\ (\#E')\ V] & \to & E'[V]
\end{array}$$

Figure 2: Operational semantics of λ_V+**exn**

$$\begin{array}{lcl}
V & ::= & x \mid n \mid \lambda x.M \mid e \\
E & ::= & [\,] \mid (E\ M) \mid (V\ E) \mid (\texttt{succ}\ E) \mid (\texttt{pred}\ E) \mid (\texttt{if0}\ E\ \texttt{then}\ M\ \texttt{else}\ M) \\
 & \mid & (\texttt{raise}\ E\ M \mid (\texttt{raise}\ V\ E) \mid (\texttt{handle}\ e\ E\ N) \mid (\texttt{handle}\ e\ V\ E)
\end{array}$$

$$\begin{array}{lcll}
E[(\lambda x.\ P)\ V] & \to & E[P[x \mapsto V]] & \\
E[\texttt{succ}\ n] & \to & E[n+1] & \\
E[\texttt{pred}\ 0] & \to & E[0] & \\
E[\texttt{pred}\ (n+1)] & \to & E[n] & \\
E[\texttt{if0}\ 0\ \texttt{then}\ M\ \texttt{else}\ N] & \to & E[M] & \\
E[\texttt{if0}\ (n+1)\ \texttt{then}\ M\ \texttt{else}\ N] & \to & E[N] & \\
E[\texttt{rec}\ f(x).\ M] & \to & E[\lambda x.M[f \mapsto \texttt{rec}\ f(x).\ M]] & \\
E[\texttt{handle}\ e\ V_h\ E_e[\texttt{raise}\ e\ V]] & \to & E[V_h V] & E_e \neq E_1[\texttt{handle}\ e\ V_{h2}\ E_2] \\
E[\texttt{handle}\ e\ V_h\ V] & \to & E[V] &
\end{array}$$

In the semantics of `callcc` that we use here, the continuations are represented as evaluation contexts E. To avoid inconsistencies, however, it is necessary to add some notation that specifies how far a reified continuation extends from the hole. We write $\#E$ for the continuation that consists of the reified evaluation context E. Note that $\#$ acts a sort of binder for the hole in the continuation: the operation of plugging the hole in an evaluation context does not affect the holes under $\#$ in any reified continuations. An alternative notation would plug the hole with a variable, and then use a binder for the variable [17], writing $\gamma x.E[x]$ instead of $\#E$. Felleisen writes continuations as $\lambda x.\mathcal{A}(E[x])$, where the $\mathcal{A}$ is an operator that discards the context of the continuations upon invocation.

As an example of how reified evaluation contexts must be annotated with $\#$, consider the following reduction which yields a continuation as a first-class value:

$$\begin{array}{rl} & \texttt{callcc}\,(\lambda k.\texttt{throw}\,(\texttt{callcc}\,(\lambda h.\texttt{throw}\;k\;h))\;42) \\ \twoheadrightarrow & \#(\texttt{throw}\,[\,]\;42) \end{array}$$

This result is a perfectly respectable continuation for a continuation, of type `int cont cont`, waiting for a continuation to be plugged into the hole, so that 42 can be thrown to it. But if we did not take care with the $\#$, we might confuse $\#(\texttt{throw}\,[\,]\;42)$ with $(\texttt{throw}\,(\#[\,])\;42)$, which reduces 42.

For exceptions, the operations are a simplified form of handling and raising, as in ML:

$$M \quad ::= \quad \ldots \mid \texttt{handle}\;e\;M\;N \mid \texttt{raise}\;M\;N$$

The operation $\texttt{raise}\;e\;V$ raises the exception e with some value V; such an exception can then be caught and handled. The syntax for this is $\texttt{handle}\;e\;M\;N$, where M will act as the handler for raising of e in the evaluation of N.

To formalize the operational semantics of exceptions, we need to refine the notion of evaluation context a little. Let E_e range over evaluation contexts that do not handle E, that is, E_e is not of the form $E_1[\texttt{handle}\;e\;V_{h2}\;E_2]$. Then the reduction for a `raise` inside a dynamically enclosing `handle` is this:

$$E[\texttt{handle}\;e\;V_h\;E_e[\texttt{raise}\;e\;V]] \quad\rightarrow\quad E[V_h V]$$

If the term inside a handler evaluates to a value, then the handler in discarded:

$$E[\texttt{handle}\;e\;V_h\;V] \quad\rightarrow\quad E[V]$$

An uncaught exception $\texttt{raise}\;e\;V$ without a dynamically enclosing handler admits no reduction.

Definition 2.2 (λ_V+exn) Let λ_V+**exn** be defined by the operational semantics rules in Figure 2.

This version of exceptions (based on the "simple exceptions" of Gunter, Rémy and Riecke [8]) differs from those in ML in that exceptions are not constructors. The fact that exceptions in Standard ML [15] 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. Furthermore, we only consider global exceptions, as in CAML [13]. Although the local declaration of exceptions as found in Standard ML adds expressive power [26], it does so in a way that is not relevant for control as such, namely by the generation of new exception names.

It is worth emphasizing in what sense exceptions are and are not first-class. Note that an exception name e can be passed around like any other value:

$$\begin{array}{ll} & (\lambda x.\texttt{handle}\ e\ (\lambda x.x)\ (\texttt{raise}\ x\ 1))\,e \\ \to & \texttt{handle}\ e\ (\lambda x.x)\ (\texttt{raise}\ e\ 1) \\ \to & (\lambda x.x)\,1 \\ \to & 1 \end{array}$$

However, while the exception name (which is little more than an atomic constant), can be passed around, the exception handler (which is an evaluation context) cannot be made into a value.

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 [20, 19], since the latter is essentially a spartan exception mechanism without handlers.

A language combining exceptions and continuations can be defined by simply merging the operational semantics for exceptions and that for continuations:

Definition 2.3 (λ_V+cont+exn) Let λ_V+**cont**+**exn** be the language defined by the union of the reduction rules of λ_V+**cont** and λ_V+**exn**.

When exceptions and continuation are combined in this way, seizing a continuation also includes the exception handler that is in force in that continuation; throwing to the continuation later thus re-installs the exception handler. Concretely, in the reduction rule for `callcc`, the exception handler, being part of the evaluation context, becomes part of the reified continuation, as in this reduction step:

$$E_1[\texttt{handle}\ e\ V\ E_e[\texttt{callcc}\,M]] \to E_1[\texttt{handle}\ e\ V\ E_e[M\,(\#E_1[\texttt{handle}\ e\ V\ E_e])]]$$

Arguably, this conforms to the intended meaning of continuation constructs in the presence of exception handlers, and to the semantics of `callcc` in Standard ML of New Jersey. It would also be possible to combine `callcc` with exceptions in such a way that capturing a continuation does not encompass the exception handler associated with it, and that throwing leaves the exception handler at the point of throw in force, rather than installing one associated with the continuation. We will return to this distinction in Section 5; the results there reinforce the view that the first semantics mentioned here is the natural one for the combination of both forms of control, exceptions and continuations.

By adding various flavours of control to a purely functional language, we defined languages that are stateless, lacking assignment and even weaker forms of state such as `gensym`; our next extension adds assignment in the form of ML-style references [15] . The grammar is extended by the following clauses:

$$M ::= \texttt{ref}\,M \mid M \texttt{:=} M \mid \texttt{!}M$$

The intended meaning is that $\texttt{ref}\,M$ creates a new assignable reference whose initial value is that of M. An assignement expression $M_1 \texttt{:=} M_2$ evaluates M_1 and assigns the value of M_2 to it. A reference M can be dereferenced by writing and $\texttt{!}\ M$, which fetches its value.

To give meaning to assignments, we add stores to the operational semantics. A store is a finite function from addresses to values. If s is a store, we write $\mathsf{dom}(s)$ for its domain of definition (that is, the addresses having a definite value in this store). We write $s + \{a \mapsto V\}$ for the store that holds the value V at address a, and whose content is otherwise that of s:

$$(s + \{a \mapsto V\})(b) = \begin{cases} V & \text{if } b = a \\ s(b) & \text{otherwise} \end{cases}$$

Figure 3: Operational semantics of state

$$
\begin{array}{rcl}
V & ::= & \ldots \mid a \\
E & ::= & \ldots \mid (\texttt{ref}\, E) \mid (\texttt{!}\; E) \mid (E\texttt{:=}M) \mid (V\texttt{:=}E)
\end{array}
$$

$$
\begin{array}{lcll}
s, E[\texttt{ref}\, V] & \to & s + \{a \mapsto V\}, E[a] & \text{where } a \notin \mathsf{dom}(s) \\
s, E[\texttt{!}\; a] & \to & s, E[s(a)] & \\
s, E[a\texttt{:=}V] & \to & s + \{a \mapsto V\}, E[V] &
\end{array}
$$

Here the address a may or may not be already in $\mathsf{dom}(s)$, that is, we could either be updating the store or allocating a new address.

For an expression M, let $\mathsf{Addr}(M)$ be the set of addresses occurring in M. The operational semantics for state works on pairs (s, P) of stores s and programs P; we usually omit the parentheses around such pairs, so that the reduction rules have the form $s, P \to s_1, Q$, where s is the old, and s_1 the new state.

To avoid listing rules that do not affect the store, we adapt the State Convention in the Definition of Standard ML: whenever there is a reduction rule $P \to Q$ in a language without state, we read this as shorthand for the rule $s, P \to s, Q$ which does not change the state.

We define a language with exceptions and state, as well as one with continuations and state:

Definition 2.4 (λ_V+cont+state) Let λ_V+**cont**+**state** be the language defined by the operational semantics in Figure 1, subject to the state convention, together with the reductions in Figure 3.

Definition 2.5 (λ_V+exn+state) Let λ_V+**exn**+**state** be the language defined by the operational semantics in Figure 2, subject to the state convention, together with the reductions in Figure 3.

In the body of this paper, we will use an untyped language. However, the results restrict to a typed subset: the operational semantics is of the type-erasure variety in that types do not have effects at run time. The typing of all language constructs, following ML, is given in Figure 4. For the typing of exceptions, it is assumed that for all exception names e there is an associated type σ_e that this exception may carry. This is comparable to the situation in CAML, where all exceptions are declared at the top level.

Definition 2.6 For the languages λ_V+**exn**, λ_V+**cont**, λ_V+**cont**+**exn**, λ_V+**cont**+**state** and λ_V+**exn**+**state**, let their *typed subset* be the language obtained by restricting them to expressions typeable according to Figure 4.

Having defined the reduction relation $\to$ for the different programming language fragments, we need some definitions that build on $\to$.

Definition 2.7 We use the following notation for reductions:

- The reflexive, transitive closure of $\to$ is written as $\twoheadrightarrow$.
- We write $P \not\twoheadrightarrow$ if there is no Q such that $P \to Q$.

Figure 4: Typing rules

$$\frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau} \qquad \frac{\Gamma \vdash M : \mathtt{int} \qquad \Gamma \vdash N_i : \tau}{\Gamma \vdash (\mathtt{if0}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2) : \tau}$$

$$\frac{\Gamma \vdash M : \mathtt{int}}{\Gamma \vdash (\mathtt{pred}\ M) : \mathtt{int}} \qquad \frac{\Gamma \vdash M : \mathtt{int}}{\Gamma \vdash (\mathtt{succ}\ M) : \mathtt{int}}$$

$$\frac{\Gamma, x : \sigma_1 \vdash M : \sigma_2}{\Gamma \vdash (\lambda x.\ M) : (\sigma_1 \to \sigma_2)} \qquad \frac{\Gamma, f : (\sigma_1 \to \sigma_2), x : \sigma_1 \vdash M : \sigma_2}{\Gamma \vdash (\mathtt{rec}\ f(x).\ M) : (\sigma_1 \to \sigma_2)}$$

$$\frac{\Gamma \vdash M : (\sigma\ \mathtt{cont} \to \sigma)}{\Gamma \vdash (\mathtt{callcc}\ M) : \sigma} \qquad \frac{\Gamma \vdash M : (\sigma\ \mathtt{cont}) \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (\mathtt{throw}\ M\ N) : \tau}$$

$$\frac{}{\Gamma \vdash e : (\mathtt{exn}\ \sigma_e)} \qquad \frac{\Gamma \vdash e : (\mathtt{exn}\ \sigma) \qquad \Gamma \vdash N : (\sigma \to \tau) \qquad \Gamma \vdash P : \tau}{\Gamma \vdash (\mathtt{handle}\ e\ N\ P) : \tau}$$

$$\frac{\Gamma \vdash M : (\mathtt{exn}\ \sigma) \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (\mathtt{raise}\ M\ N) : \tau} \qquad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash (\mathtt{ref}\ M) : \tau\,\mathtt{ref}}$$

$$\frac{\Gamma \vdash M : \tau\,\mathtt{ref}}{\Gamma \vdash\ !\,M : \tau} \qquad \frac{\Gamma \vdash M : \tau\,\mathtt{ref} \qquad \Gamma \vdash N : \tau}{\Gamma \vdash M \mathrel{\mathtt{:=}} N : \tau}$$

- We say P is *stuck* if $P \not\rightarrow$, but P it is neither a value nor an uncaught exception $E_e[\mathtt{raise}\ e\ V]$.
- We write $P \downharpoonleft Q$ iff $P \twoheadrightarrow Q$ and $Q \not\rightarrow$. We write $P\downharpoonleft$ if $P \downharpoonleft Q$ for some Q.
- We write $P \uparrow$ iff there is no Q such that $P \downharpoonleft Q$; we say P *diverges.*
- For an expression M and an evaluation context E, we write $M \Lsh E$ if there is no value V such that $M \twoheadrightarrow E[V]$. If $\mathcal{C}$ is a set of evaluation contexts, we write $M \Lsh \mathcal{C}$ if $M \Lsh E$ for all $E \in \mathcal{C}$.

Notice that $P\downharpoonleft$ is not the same as successful termination, as it also includes stopping with an uncaught exception, and "going wrong" due to a run-time type error. The relation $\downharpoonleft$ is related to the big-step semantics for exceptions used in the Definition of Standard ML [15] and in the predecessor paper [17] in that $P \downharpoonleft E_e[\mathtt{raise}\ e\ V]$ iff $P \downharpoonleft \mathtt{raise}\ e\ V$. The stripping-off of evaluation contexts is implicit in the derivation rules of the big-step semantics (due to the exception convention), while the small-step semantics does this in the reduction of `handle`.

We will use the relation $\Lsh$, which specifies that a program never invokes a given continuation, in reasoning about first-class continuations. Note that, if $M \uparrow$, then $E[M] \Lsh E$. In a typed language without control, the converse holds as well (assuming that the typing rules out that M gets stuck). But in the presence of exceptions, it could also be the case that $M \twoheadrightarrow E_e[\mathtt{raise}\ e\ V]$. In a language with first-class continuation, it could be the case that $M \twoheadrightarrow \mathtt{throw}\ \#E'\ V$ for some different $E' \neq E$; but then it could still happen that $M \twoheadrightarrow \mathtt{throw}\ \#E\ V$, so it may be quite difficult to decide whether $M \Lsh E$ holds or not.

Definition 2.8 For a finite reduction of the form

$$P_0 \to P_1 \to \cdots \to P_n$$

let its *length* be n and its *size* be the sum of the sizes of P_0, P_1, ... ,P_n.

Definition 2.9 (Contextual equivalence) Two terms P and P' are *contextually equivalent* iff for all contexts C such that both $C[P]$ and $C[P']$ are closed, we have: $C[P] \twoheadrightarrow n$ for some integer n iff $C[P'] \twoheadrightarrow n$.

One could also define contextual equivalence by observing only termination, that is, by stipulating that $C[P] \downharpoonleft$ iff $C[P'] \downharpoonleft$. Observing integers rather than termination has the advantage that our results restrict even to the simply typed subset without recursion, where everything terminates.

In all cases where we show that some contextual equivalence does not hold in the presence of some constructs, we will actually show something stronger than contextual non-equivalence.

Definition 2.10 (Separation) Two terms P and P' *can be separated* iff there is a context C such that: $C[P] \twoheadrightarrow n$ for some integer n, and $C[P'] \twoheadrightarrow n'$ with $n \neq n'$.

Separability is a stronger notion than contextual in-equivalence: not only do we get an observation in one case, and possibly no observation (due to divergence) in the other; rather, we can observe the difference. Once we have a separation context, it is a matter of straightforward evaluation to verify the separation. Moreover, this can be machine-verified, so to speak, by simply feeding the resulting expression to Scheme or ML. So to show separation we will in some cases feel free just to display the required ML code.

What we regard as the essence of exceptions is that they are a form of jumps whose target is determined *dynamically*. This fundamental feature is shared by exceptions in the sense of ML [15] or Java [6], and also by dynamic control in the sense of most Lisps, such as `catch` and `throw` in Common Lisp [19] and Emacs Lisp. (Oddly, EuLisp calls such a construct `letcc`, as if it were the binding-form analogue of `call/cc` in Scheme, which it is not.)

One could even have used the greatly simplified dynamic control from [27], which involves an operator `go` that jumps to the nearest dynamically enclosing `here`. In operational semantics, we would write this as:

$$E[\mathtt{here}\, E_h[\mathtt{go}\, V]] \quad \rightarrow \quad E[V] \qquad E_h \neq E_1[\mathtt{here}\, E_2]$$

By contrast, we do not view the typing of exceptions by way of a universal type in Standard ML as particularly relevant (although it can be seen as evidence that a typing as elegant and canonical as that for continuations [3] is out of reach for exceptions).

Lillibridge has observed that due to this universal type, one can encode recursion in a simply-typed fragment of ML without recursive functions, something that is not possible with continuation operations, as they do not involve new recursive types. In that sense, exceptions were found to be "more powerful than `call/cc`" [14]. If the language already has recursion, or is untyped, thereby opening the door for **Y**-combinators, this argument is of course inapplicable. One should also be aware that 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", as was pointed out by its designers [1].

3 Exceptions cannot express continuations

We prove contextual equivalences by establishing a simulation relation. This relation must be such that the two expressions we want to show equivalent, if slotted into any context, yield related programs. Moreover, only observable-s must be related to observable-s. One then shows that the relation is preserved by the operational semantics.

In all such proofs, there is a certain amount of housekeeping, as multiple copies of the two expressions may be propagated throughout the programs. In addition, the proof becomes more difficult if effects of indefinite extent are added to the language, as we need to keep track of these.

Exceptions are relatively easy to deal with. The extent of a handler is not indefinite, and is in fact contained inside a sub-derivation of the big-step semantics. Housekeeping aside, the heart of the proof is an straightforward argument by cases: either an expression yields a value, $M \twoheadrightarrow V$, or it raises an exception, $M \twoheadrightarrow E_e[\mathtt{raise}\ e\ V]$.

Consider these two expressions:

$$(\lambda f.(f\,T; f\,F))M$$

and

$$(\lambda f g.(f\,T; g\,F))M M$$

For a language with exceptions, we can argue by cases. Either M raises an exception, in which case both expressions raise an exception; or M returns a value V. In the absence of state, both copies of M in the second expression must evaluate to the same V. So both expression will behave like $(V\,T; V\,F)$.

Lemma 3.1 Let M be an expression and E an evaluation context of λ_V+**exn**. If $M \twoheadrightarrow Q$, then $E[M] \twoheadrightarrow E[Q]$, where the latter reduction is no smaller than the former.

Note that this only hold for $\twoheadrightarrow$, but not for $\downharpoonright$, as E may catch exceptions raised in M.

Lemma 3.2 If $E[M]\downharpoonright$, then $M\downharpoonright$.

Lemma 3.3 Let M be an expression and E an evaluation context of λ_V+**exn**. If M is stuck, then $E[M]$ is stuck.

Our proof technique relates expressions by a simulation relation, written as $M \sim M'$. Since the operational semantics for exceptions factors out an evaluation context, we also need to relate evaluation contexts. This relation is also written as $E \approx E'$. The relation between evaluation contexts depends on that for expressions: for instance, $42 \approx 42$, hence also $[\ ]\,42 \approx [\ ]\,42$: the evaluation context that applies something to 42 is equivalent to itself.

Definition 3.4 Let the relations $\approx$ between expressions, respectively between evaluation contexts, of λ_V+**exn** be defined as in Figures 5 and 6.

Lemma 3.5 Let M be an expression and C a context of λ_V+**exn** such that $C[(\lambda f.(f\,T; f; F))\,M]$ is closed. Then

$$C[(\lambda f.(f\,T; f; F))\,M] \approx C[(\lambda f g.(f\,T; g\,F))\,M\,M]$$

In particular, Lemma A.3 implies that $\approx$ relates values to values.

The proof of contextual equivalence for the language with exceptions is reasonably straightforward. We reason about the behaviour of a program P in terms of a reduction of the form $P \twoheadrightarrow Q$ where $Q \not\twoheadrightarrow$. Because exceptions, unlike continuations, do not have indefinite extent, all control behaviour of a terminating program P is contained in a reduction of limited length.

Lemma 3.6 (One-step simulation) Let P and P' be closed expressions of λ_V+**exn** such that $P \approx P'$. Then at least one of the following is the case:

Figure 5: Simulation relation on expressions

$$\frac{M \approx M'}{\texttt{raise}\ e\ M \approx \texttt{raise}\ e\ M} \qquad \frac{M \approx M' \qquad N \approx N'}{\texttt{handle}\ e\ M\ N \approx \texttt{handle}\ e\ M'\ N'}$$

$$\frac{M \approx M' \qquad N \approx N'}{MN \approx M'N'} \qquad \frac{}{e \approx e}$$

$$\frac{M \approx M' \qquad N \approx N' \qquad L \approx L'}{\texttt{if0}\ M\ \texttt{then}\ N\ \texttt{else}\ L \approx \texttt{if0}\ M'\ \texttt{then}\ N'\ \texttt{else}\ L'} \qquad \frac{}{n \approx n}$$

$$\frac{M \approx M'}{\lambda x.M \approx \lambda x.M'} \qquad \frac{M \approx M'}{\texttt{rec}\ f(x).\ M \approx \texttt{rec}\ f(x).\ M'}$$

$$\frac{M \approx M'}{\texttt{pred}\ M \approx \texttt{pred}\ M'} \qquad \frac{M \approx M'}{\texttt{succ}\ M \approx \texttt{succ}\ M'}$$

$$\frac{M \approx M'}{(\lambda f.(f\,T; f\,F))\,M \approx (\lambda f g.(f\,T; g\,F))M'M'} \qquad \frac{}{x \approx x}$$

Figure 6: Simulation relation on evaluation contexts

$$\frac{E \approx E' \qquad N \approx N'}{EN \approx E'N'} \qquad \frac{V \approx V' \qquad E \approx E'}{VE \approx V'E'}$$

$$\frac{E \approx E' \qquad N \approx N' \qquad L \approx L'}{\texttt{if0}\ E\ \texttt{then}\ N\ \texttt{else}\ L \approx \texttt{if0}\ E'\ \texttt{then}\ N'\ \texttt{else}\ L'} \qquad \frac{}{[\,] \approx [\,]}$$

$$\frac{E \approx E' \qquad N \approx N'}{\texttt{raise}\ E\ N \approx \texttt{raise}\ E'\ N'} \qquad \frac{V \approx V' \qquad E \approx E'}{\texttt{raise}\ V\ E \approx \texttt{raise}\ V'\ E'}$$

$$\frac{E \approx E' \qquad M \approx M'}{\texttt{handle}\ e\ E\ M \approx \texttt{handle}\ e\ E'\ M'} \qquad \frac{E \approx E'}{\texttt{succ}\ E \approx \texttt{succ}\ E'}$$

$$\frac{V \approx V' \qquad E \approx E'}{\texttt{handle}\ e\ V\ E \approx \texttt{handle}\ e\ V'\ E'} \qquad \frac{E \approx E'}{\texttt{pred}\ E \approx \texttt{pred}\ E'}$$

1. P and P' are both values.

2. P and P' are both stuck.

3. $P \to P_1$ and $P' \to P'_1$ such that $P_1 \approx P'_1$.

4. $P = E_e[\texttt{raise}\ e\ V]$ and $P' = E'_e[\texttt{raise}\ e\ V']$ with $V \approx V'$.

5. $P = E[(\lambda f.(fT; fF))M]$ and $P' = E'[(\lambda fg.(f\,T; g\,F))M'M']$.

Lemma 3.7 (Simulation) Let P and P' be closed expressions of λ_V+**exn** such that $P \approx P'$. Then:

- If $P \downharpoonleft Q$, then $P' \downharpoonleft Q'$ for some Q'. Furthermore, if Q is a value, then $Q \approx Q'$; if Q is an uncaught exception $Q = E_e[\texttt{raise}\ e\ V]$, then $Q' = E'_e[\texttt{raise}\ e\ V']$ with $V \approx V'$.

- Symmetrically, if $P' \downharpoonleft Q'$, then $P \downharpoonleft Q$ for some Q. Furthermore, if Q' is a value, then $Q \approx Q'$; if Q' is an uncaught exception $Q' = E'_e[\texttt{raise}\ e\ V']$, then $Q = E_e[\texttt{raise}\ e\ V]$ with $V \approx V'$.

Proof We deal only with the first statement, as the other is symmetric. The proof is by induction on the size of the reduction. We assume $P \downharpoonleft Q$, and that the statement of the lemma is true for reductions of smaller size. The cases that we need to consider are given by Lemma 3.6. The most important case is this:

$$\begin{array}{rcl} P & = & E[(\lambda f.(f\,T; f\,F))M] \\ P' & = & E'[(\lambda fg.(f\,T; g\,F))M'M'] \end{array}$$

with $E \approx E'$ and $M \approx M'$.

By Lemma 3.2, $P\downharpoonleft$ implies that $M \downharpoonleft N$ for some N. We proceed by cases on what N is: a value, an uncaught exception or stuck.

1. $M \downharpoonleft V$ for some value V. Then we have by Lemma 3.1 (applied to M and the evaluation context $E[(\lambda f.(f\,T; f\,F))[\,]]$):

$$\begin{array}{rcl} P & \twoheadrightarrow & E[(\lambda f.(f\,T; f\,F))M] \\ & \twoheadrightarrow & E[(\lambda f.f\,T; f\,F)V] \\ & \to & E[V\,T; V\,F] \end{array}$$

As $M \approx M'$, and $M \downharpoonleft N$ by a smaller reduction, we can apply the induction hypothesis; hence $M' \downharpoonleft V'$ for some value V' such that $V \approx V'$. Therefore, for P' we have by Lemma 3.1 (applied to M' with the the evaluation contexts $E'[(\lambda fg.(f\,T; g\,F))[\,]M']$ and $E[(\lambda g.(V'\,T; g\,F))[\,]]$):

$$\begin{array}{lll} & E'[(\lambda fg.(f\,T; g\,F))M'M'] & \\ \twoheadrightarrow & E'[(\lambda fg.(f\,T; g\,F))V'M'] & \text{(by Lemma 3.1)} \\ \to & E'[(\lambda g.(V'\,T; g\,F))M'] & \\ \twoheadrightarrow & E'[(\lambda g.(V'\,T; g\,F))V'] & \text{(by Lemma 3.1)} \\ \to & E'[V'\,T; V'\,F] & \end{array}$$

Because $E \approx E'$ and $V \approx V'$, $E[V\,T; V\,F] \approx E'[V'\,T; V'\,F]$. Moreover, $E[V\,T; V\,F] \downharpoonleft Q$ by a reduction smaller than the one for $P \downharpoonleft Q$. So by the induction hypothesis, we have $E'[V'\,T; V'\,F] \downharpoonleft Q'$, where Q' fulfills the additional conditions.

2. M raises an exception: $M \downharpoonleft E_e[\texttt{raise}\ e\ V]$. There are two subcases, depending on whether this exception is caught by E or not.

   (a) The exception is not caught, that is, E is of the form E_{e2}. Then, again by Lemma 3.1:

   $$\begin{array}{rcl} P & \twoheadrightarrow & E_{e2}[(\lambda f.(f\,T; f\,F))M] \\ & \twoheadrightarrow & E_{e2}[(\lambda f.(f\,T; f\,F))E_e[\texttt{raise}\ e\ V]] = Q \end{array}$$

   Note that the $\texttt{raise}\ e\ V$ is in evaluation context position in Q. Hence $Q \not\twoheadrightarrow$. As $M \downharpoonleft E_e[\texttt{raise}\ e\ V]$ by a smaller reduction than that of $P \downharpoonleft Q$, we can apply the induction hypothesis to it. Thus, $M' \downharpoonleft E'_e[\texttt{raise}\ e\ V']$, with $V \approx V'$. Therefore, we have for P':

   $$\begin{array}{rcl} P' & \twoheadrightarrow & E'_{e2}[(\lambda fg.(f\,T; g\,F))M'] \\ & \twoheadrightarrow & E'_{e2}[(\lambda fg.(f\,T; g\,F))E'_e[\texttt{raise}\ e\ V']M'] =: Q' \end{array}$$

   For Q and Q', we have $V \approx V'$, as required.

   (b) E is of the form $E = E_1[\texttt{handle}\ e\ V_h\ E_{e2}]$. Then we have for P:

   $$\begin{array}{rcl} P & \twoheadrightarrow & E_1[\texttt{handle}\ e\ V_h\ E_{e2}[(\lambda f.(f\,T; f\,F))M]] \\ & \twoheadrightarrow & E_1[\texttt{handle}\ e\ V_h\ E_{e2}[(\lambda f.(f\,T; f\,F))E_e[\texttt{raise}\ e\ V]]] \\ & \to & E_1[V_h\,V] \end{array}$$

   Applying the induction hypothesis to the smaller reductions $M \downharpoonleft E_e[\texttt{raise}\ e\ V]$, we have for P'

   $$\begin{array}{rcl} P' & \twoheadrightarrow & E'_1[\texttt{handle}\ e\ V'_h\ E'_{e2}[(\lambda fg.(f\,T; g\,F))M'M']] \\ & \twoheadrightarrow & E'_1[\texttt{handle}\ e\ V'_h\ E'_{e2}[(\lambda fg.(fT; gF))E'_e[\texttt{raise}\ e\ V']M']]] \\ & \to & E'_1[V'_h\,V'] \end{array}$$

   The statement again follows by applying the induction hypothesis to the smaller reduction $E_1[V_2\,V] \downharpoonleft Q$.

3. $M \downharpoonleft N$ and N is stuck. Then

   $$P \twoheadrightarrow E[(\lambda f.(f\,T; f\,F))\,N]$$

   where the latter term is also stuck. By the induction hypothesis applied to $M\downharpoonleft$, we have that $M' \downharpoonleft N'$ where N' is stuck. Hence

   $$P' \twoheadrightarrow E'[(\lambda fg.f\,T; g\,F)N'M']$$

   where the latter term is stuck.

□

With continuations, however, we can easily separate expressions like these. (Similar reductions are also discussed in [25].)

Lemma 3.8 The expressions $\lambda f.(f\,T; f\,F))\,M$ and $(\lambda fg.(f\,T; g\,F))\,M\,M$ can be separated by continuations.

Proof Let $M = \texttt{callcc}\,(\lambda k.\lambda x.\texttt{throw}\,k\ (\lambda y.x))$ and $C = [\,]$ Then

$$\begin{array}{rcl} (\lambda f.(fT; fF))M & \twoheadrightarrow & T \\ (\lambda fg.(f\,T; g\,F))MM & \twoheadrightarrow & F \end{array}$$

It is worth to see in a little more detail how $(\lambda f.(fT; fF))M$ manages to return T:

$$\begin{array}{rl} & (\lambda f.(f\,T; f\,F))\,(\texttt{callcc}\,(\lambda k.\lambda x.\texttt{throw}\,k\ (\lambda y.x))) \\ \twoheadrightarrow & (\lambda f.(f\,T; f\,F))\,(\lambda x.\texttt{throw}\ (\#((\lambda f.(f\,T; f\,F))\,[\,])\ (\lambda y.x)) \\ \to & (\lambda x.\texttt{throw}\ (\#((\lambda f.(f\,T; f\,F))\,[\,])\ (\lambda y.x))\,T; \\ & (\lambda x.\texttt{throw}\ (\#((\lambda f.(f\,T; f\,F))\,[\,])\ (\lambda y.x))\,F \\ \to & \texttt{throw}\ (\#((\lambda f.(f\,T; f\,F))\,[\,])\ (\lambda y.T)); \\ & (\lambda x.\texttt{throw}\ (\#((\lambda f.(f\,T; f\,F))\,[\,])\ (\lambda y.x))\,F \\ \to & (\lambda f.(f\,T; f\,F))\,(\lambda y.T) \\ \twoheadrightarrow & T \end{array}$$

□

Theorem 3.9 There are expressions which are contextually equivalent in λ_V+**exn**, but which can be separated in λ_V+**cont**. This result restricts to the typed subset, and holds whether or not recursion is present.

Proof Let C be a closing context of λ_V+**exn**. Suppose that for some value V:

$$C[(\lambda f.(f\,T; f; F))\,M] \twoheadrightarrow V$$

By Lemma 3.5, we have

$$C[(\lambda f.(f\,T; f; F))\,M] \approx C[(\lambda fg.(f\,T; g\,F))\,M\,M]$$

which by Lemma 3.7 implies that

$$C[(\lambda fg.(f\,T; g\,F))\,M,\ M] \twoheadrightarrow V'$$

where $V \approx V'$; hence V' is also a value by Lemma 3.6. The converse is symmetric.

On the other hand, the two expressions can be separated with continuations by Lemma 3.8.

□

A consequence of this non-equivalence is the following:

Corollary 3.10 Exceptions cannot macro-express continuations.

Proof By Theorem 3.9 and Felleisen [4]. □

In the conference version of this paper, we used a big-step semantics for exceptions, which was shown equivalent to the small-step one, for this proof. In a big-step semantics for exceptions, the control behaviour is automatically confined inside a single derivation $P \Downarrow Q$ where $Q = V$ or $Q = \texttt{raise}\ e\ V$. While it is thus more immediately obvious how to organize the induction for a big-step semantics, this does not mean that there is any real obstacle in using a small-step semantics. We need only to keep the condition that Q does not do any further steps in the induction hypothesis $P \xrightarrow{n} Q$. We can confine reasoning about any exception handler to the finitely many reduction steps that the program spend in the lifetime of the handler.

4 Continuations without state cannot express exceptions

In this section, we will show that exceptions can break a contextual equivalence that holds in the presence of continuations. This result holds only in the absence of state, but is still somewhat surprising, given that first-class continuations seem so much more powerful than any other form of control. Intuitively the reason for the difference is that exceptions can detect parts of the dynamically enclosing evaluation context by means of handlers contained inside it. In a language with only continuations, that is impossible: even though an evaluation context may be passed around as a first-class continuation, the only way to tell something about it is to invoke it with some value. Informally speaking, if a continuation is never invoked, it might as well be any other for all we can tell about it. The proof technique is in fact largely a formalization of that ideas.

A difficulty that arises immediately is the following: how can we tell if some continuation is ever invoked or not? For exceptions it was not so difficult to find out whether a value V would ever be returned to E in $E[M]$; we could just look at M and see whether $M \twoheadrightarrow V$, in which case $E[M] \twoheadrightarrow E[V]$. No such simple argument works for continuations; the principle that $M \twoheadrightarrow V$ implies $E[M] \twoheadrightarrow E[V]$ is of course violated by $M = \texttt{callcc}\,(\lambda k.N)$, in which case the value V may depend on E, and who knows what N will do with the reified $\#E$. The central idea in our proof technique is that we avoid the burden of proving if a continuation is ever invoked or not. Rather, we deal with each case separately, and since by excluded middle one or the other must be the case, that is enough for the proof. In particular, in the case that the continuation is not invoked, we gain a very strong assurance about the future behaviour of the expression that would be very difficult to prove otherwise.

The following equivalence can be broken by exceptions, but not by continuations:

$$(\lambda f.M)(\lambda x.O) \cong (\lambda f.M)(\lambda y.((\lambda f.M)(\lambda x.O); O))$$

where y is not free in M.

Notice that $(\lambda f.M)$ on the left-hand side and $(\lambda f.M)$ on the right hand side are given the same argument $(\lambda x.O)$, so that M cannot find out any difference by looking at this argument. However, the second M on the right will be called inside the dynamic extent of the call of the first copy of M, which can be detected using the exception mechanism.

Continuations, by contrast, cannot detect the presence a second copy of M. Intuitively, we need to show that $(\lambda f.M)(\lambda x.O)$ can be simulated (possibly in more steps) by

$$(\lambda f.M)(\lambda y.((\lambda f.M)(\lambda x.O); O))$$

During the evaluation of the latter expression, we may be either in the first M, the caller, or in the second M, the callee. In the evaluation of the caller, the argument is different (comprising the callee) and the continuation the same; conversely, in the evaluation of the callee, the argument is the same and the continuation is different (containing parts of the caller). What we need to construct is a simulation that stays inside the caller if the callee returns to the call, but switches to the callee if it does not.

The caller cannot pass anything to the callee because y is not free in M; and the callee cannot return anything to the caller because the return value is always 0. The point immediately in front of the semicolon is the linchpin of the proof, for the argument turns on whether a value is eventually returned to this point or not. More formally, this program point is represented as the continuation $E[[\,]; O]$ if $(\lambda y.(\lambda f.M)(\lambda x.O); O)$ is called with return continuation E. For exceptions the question whether the expression would return a value or raise an exception was open during the evaluation of the expression. The continuation has indefinite extent, so the question can stay open indefinitely.

Figure 7: Pre-simulation relations on expressions

$$\frac{M \approx^{\mathcal{A}}_{\mathcal{C}} M' \qquad N \approx^{\mathcal{A}}_{\mathcal{C}} N'}{MN \approx^{\mathcal{A}}_{\mathcal{C}} M'N'} \qquad \frac{M \approx^{\mathcal{A}}_{\mathcal{C}} M' \qquad N \approx^{\mathcal{A}}_{\mathcal{C}} N'}{\texttt{throw}\, M\; N \approx^{\mathcal{A}}_{\mathcal{C}} \texttt{throw}\, M'\; N'}$$

$$\frac{M \approx^{\mathcal{A}}_{\mathcal{C}} M' \qquad N \approx^{\mathcal{A}}_{\mathcal{C}} N' \qquad L \approx^{\mathcal{A}}_{\mathcal{C}} L'}{\texttt{if0}\, M\; \texttt{then}\, N\; \texttt{else}\, L \approx^{\mathcal{A}}_{\mathcal{C}} \texttt{if0}\, M'\; \texttt{then}\, N'\; \texttt{else}\, L'} \qquad \frac{}{n \approx^{\mathcal{A}}_{\mathcal{C}} n}$$

$$\frac{M \approx^{\mathcal{A}}_{\mathcal{C}} M'}{\lambda x.M \approx^{\mathcal{A}}_{\mathcal{C}} \lambda x.M'} \qquad \frac{M \approx^{\mathcal{A}}_{\mathcal{C}} M'}{\texttt{callcc}\, M \approx^{\mathcal{A}}_{\mathcal{C}} \texttt{callcc}\, M'}$$

$$\frac{M \approx^{\mathcal{A}}_{\mathcal{C}} M'}{\texttt{rec}\; f(x).\, M \approx^{\mathcal{A}}_{\mathcal{C}} \texttt{rec}\; f(x).\, M'} \qquad \frac{E \sim^{\mathcal{A}}_{\mathcal{C}} E'}{\#E \approx^{\mathcal{A}}_{\mathcal{C}} \#E'}$$

$$\frac{M \approx^{\mathcal{A}}_{\mathcal{C}} M'}{\texttt{pred}\, M \approx^{\mathcal{A}}_{\mathcal{C}} \texttt{pred}\, M'} \qquad \frac{M \approx^{\mathcal{A}}_{\mathcal{C}} M'}{\texttt{succ}\, M \approx^{\mathcal{A}}_{\mathcal{C}} \texttt{succ}\, M'}$$

$$\frac{M' \in \mathcal{A}}{(\lambda x.0) \approx^{\mathcal{A}}_{\mathcal{C}} (\lambda y.((\lambda f.M')(\lambda x.O); O))} \qquad \frac{}{x \approx^{\mathcal{A}}_{\mathcal{C}} x}$$

$$\frac{M \approx^{\mathcal{A}}_{\mathcal{C}} M'}{(\lambda f.M)(\lambda x.O) \approx^{\mathcal{A}}_{\mathcal{C}} (\lambda f.M')(\lambda y.((\lambda f.M')(\lambda x.O); O))}$$

$$\frac{M \approx^{\mathcal{A}}_{\mathcal{C}} M' \qquad E' \in \mathcal{C}}{E[M] \sim^{\mathcal{A}}_{\mathcal{C}} E'[M']} \qquad \frac{M \approx^{\mathcal{A}}_{\mathcal{C}} M'}{M \sim^{\mathcal{A}}_{\mathcal{C}} M'}$$

The equivalence is true even in the presence of upward continuations, that is M could seize the current continuation and pass it to the outside, as in $M = \texttt{callcc}\,(\lambda k.\texttt{throw}\; h\; k)$.

Then definition of a suitable simulation relation involves some relatively complex induction. Because we have first-class continuations in the language, evaluation contexts can become part of expressions, which may themselves become part of other evaluation contexts. We first define relations that we call *pre*-simulations because they do not entail observational equivalence without some further assumptions.

Definition 4.1 (Pre-simulation relations) Let $\mathcal{C}$ be a set of closed evaluation contexts and $\mathcal{A}$ a set of closed expressions of λ_V+**cont**. We define relations $\sim^{\mathcal{A}}_{\mathcal{C}}$ and $\approx^{\mathcal{A}}_{\mathcal{C}}$ both between expression and evaluation contexts, by mutual recursion as given in Figures 7 and 8.

Roughly speaking, these definitions mean that if $P \approx^{\mathcal{A}}_{\mathcal{C}} P'$, then positions in P and P' may differ in functions (provided they appear in $\mathcal{A}$) and in reified continuations (provided they appear in $\mathcal{C}$). If $P \sim^{\mathcal{A}}_{\mathcal{C}} P'$, then the two terms may also differ in a prefix of the *current*, not reified, continuation (provided it appears in $\mathcal{C}$). The definitions for evaluation contexts are analogous to those for expressions.

The point of the relation $\sim^{\mathcal{A}}_{\mathcal{C}}$ is the following lemma, a stepping-stone towards a simulation:

Lemma 4.2 (One-step simulation) Let P and P' be closed expressions of λ_V+**cont** with $P \sim^{\mathcal{A}}_{\mathcal{C}} P'$. Then at least one of the following is the case:

1. $P \not\twoheadrightarrow$ and $P' \not\twoheadrightarrow$.

2. $P \rightarrow P_1$ and $P' \rightarrow P_1'$ with $P_1 \sim^{\mathcal{A}}_{\mathcal{C}} P_1'$.

Figure 8: Pre-simulation relations on evaluation contexts

$$\frac{E \approx^{\mathcal{A}}_{\mathcal{C}} E' \qquad N \approx^{\mathcal{A}}_{\mathcal{C}} N'}{EN \approx^{\mathcal{A}}_{\mathcal{C}} E'N'} \qquad \frac{V \approx^{\mathcal{A}}_{\mathcal{C}} V' \qquad E \approx^{\mathcal{A}}_{\mathcal{C}} E'}{VE \approx^{\mathcal{A}}_{\mathcal{C}} V'E'}$$

$$\frac{E \approx^{\mathcal{A}}_{\mathcal{C}} E' \qquad N \approx^{\mathcal{A}}_{\mathcal{C}} N'}{\texttt{throw}\; E\; N \approx^{\mathcal{A}}_{\mathcal{C}} \texttt{throw}\; E'\; N'} \qquad \frac{V \approx^{\mathcal{A}}_{\mathcal{C}} V' \qquad E \approx^{\mathcal{A}}_{\mathcal{C}} E'}{\texttt{throw}\; V\; E \approx^{\mathcal{A}}_{\mathcal{C}} \texttt{throw}\; V'\; E'}$$

$$\frac{E \approx^{\mathcal{A}}_{\mathcal{C}} E' \qquad N \approx^{\mathcal{A}}_{\mathcal{C}} N' \qquad L \approx^{\mathcal{A}}_{\mathcal{C}} L'}{\texttt{if0}\; E\; \texttt{then}\; N\; \texttt{else}\; L \approx^{\mathcal{A}}_{\mathcal{C}} \texttt{if0}\; E'\; \texttt{then}\; N'\; \texttt{else}\; L'} \qquad \frac{}{[\,] \approx^{\mathcal{A}}_{\mathcal{C}} [\,]}$$

$$\frac{E \approx^{\mathcal{A}}_{\mathcal{C}} E'}{\texttt{pred}\; E \approx^{\mathcal{A}}_{\mathcal{C}} \texttt{pred}\; E'} \qquad \frac{E \approx^{\mathcal{A}}_{\mathcal{C}} E'}{\texttt{succ}\; E \approx^{\mathcal{A}}_{\mathcal{C}} \texttt{succ}\; E'}$$

$$\frac{E \approx^{\mathcal{A}}_{\mathcal{C}} E'}{E \sim^{\mathcal{A}}_{\mathcal{C}} E'} \qquad \frac{E \approx^{\mathcal{A}}_{\mathcal{C}} E' \qquad E'_1 \in \mathcal{C}}{E_1[E] \sim^{\mathcal{A}}_{\mathcal{C}} E'_1[E']}$$

3. $P = E_0[(\lambda f.N)(\lambda x.O)]$ and $P' = E'_0[(\lambda f.N')(\lambda y.((\lambda f.N')(\lambda x.O); O))]$ with $N \approx^{\mathcal{A}}_{\mathcal{C}} N'$ and $E_0 \sim^{\mathcal{A}}_{\mathcal{C}} E'_0$.
4. $P = E_0[(\lambda x.O)V]$ and $P' = E'_0[(\lambda y.((\lambda f.N')(\lambda x.O); O))V']$ with $N' \in \mathcal{A}$ and $E_0 \sim^{\mathcal{A}}_{\mathcal{C}} E'_0$.
5. $P = E_0[V]$ and $P' = E'[V']$ with $E' \in \mathcal{C}$.

Lemma 4.2 establishes that expressions related by $\sim^{\mathcal{A}}_{\mathcal{C}}$ behave the same as long as they do not touch those positions in which they differ, in the way spelled out by the last three cases.

We refine our pre-simulation relation into a proper simulation relation by requiring that expressions may only differ in continuations that will never be invoked or in functions that always return. Expressions related by $\overset{\twoheadrightarrow}{\sim}$ will then be shown to be indistinguishable.

Definition 4.3 (Simulation relation) Let the relation $\overset{\twoheadrightarrow}{\sim}$ between expressions of λ_V+**exn** be defined as follows: $P \overset{\twoheadrightarrow}{\sim} P'$ iff $P \sim^{\mathcal{A}}_{\mathcal{C}} P'$ for some $\mathcal{A}$ and $\mathcal{C}$, and moreover:

- for all $E \in \mathcal{C}$, $P' \not\twoheadrightarrow E[V]$ for any value V, and
- for all $M' \in \mathcal{A}$, if
$$P' \twoheadrightarrow E[(\lambda f.M')(\lambda x.O)]$$
for some evaluation context E, then there is a value V such that
$$E[(\lambda f.M')(\lambda x.O)] \twoheadrightarrow E[V]$$

These additional assumptions about reductions are enough to guarantee that expressions related by $\overset{\twoheadrightarrow}{\sim}$ behave the same.

Tracking continuations along the reduction could mean a forbidding amount of extra bookkeeping. We use classical logic by appealing to the excluded middle in order to avoid backtracking in the proof. While in principle there need not be anything non-constructive about the proof, classical logic internalizes backtracking in the sense that proof by contradiction is the manifestation of backtracking at the meta-level.

Lemma 4.4 (Simulation) Let P and P' be closed expressions of λ_V+**cont** with $P \overset{\rightarrow}{\sim} P'$. If $P \to Q$, then there is a Q' such that $P' \xrightarrow{+} Q'$ and $Q \overset{\rightarrow}{\sim} Q'$. If on the other hand $P \not\twoheadrightarrow$, then $P' \not\twoheadrightarrow$.

Proof Assume $P \overset{\rightarrow}{\sim} P'$. Hence $P \sim_{\mathcal{C}}^{\mathcal{A}} P'$ for some $\mathcal{A}$ and $\mathcal{C}$. By Lemma 4.2, we need only consider the following cases.

1. $P \not\twoheadrightarrow$ and $P' \not\twoheadrightarrow$. We are done immediately.
2. $P \to P_1$, $P' \to P_1'$ with $P_1 \sim_{\mathcal{C}}^{\mathcal{A}} P_1'$. Again we are done.
3. Now suppose that

$$\begin{array}{rcl} P & = & E_0[(\lambda f.M)(\lambda x.O)] \\ P' & = & E_0'[(\lambda f.M')(\lambda y.((\lambda f.M')(\lambda x.O);O))] \end{array}$$

with $E_0 \sim_{\mathcal{C}}^{\mathcal{A}} E_0'$ and $M \approx_{\mathcal{C}}^{\mathcal{A}} M'$. Both sides begin to evaluate the call:

$$\begin{array}{lll} P & \to Q & := E_0[M[f \mapsto \lambda x.O]] \\ P' & \to P_1' & := E_0'[M'[f \mapsto (\lambda y.((\lambda f.M')(\lambda x.O);O))]] \end{array}$$

Depending on the future behaviour of P_1', there are two possible cases:

(a) Whenever $P_1' \twoheadrightarrow E'[(\lambda f.M')(\lambda x.O)]$ for some E', then $E'[(\lambda f.M')(\lambda x.O)] \twoheadrightarrow E'[V']$. By definition $(\lambda x.O) \approx_{\mathcal{C}}^{\mathcal{A}\cup\{M'\}} (\lambda y.((\lambda f.M')(\lambda x.O);O))$.

$$\begin{array}{rcll} M & \approx_{\mathcal{C}}^{\mathcal{A}} & M' & \\ M & \approx_{\mathcal{C}}^{\mathcal{A}\cup\{M'\}} & M' & \text{by Lemma A.6} \\ M[f \mapsto \lambda x.O] & \approx_{\mathcal{C}}^{\mathcal{A}\cup\{M'\}} & M'[f \mapsto (\lambda y.((\lambda f.M')(\lambda x.O);O))] & \text{by Lemma A.5} \\ E_0[M[f \mapsto \lambda x.O]] & \sim_{\mathcal{C}}^{\mathcal{A}\cup\{M'\}} & E_0'[M'[f \mapsto (\lambda y.((\lambda f.M')(\lambda x.O);O))]] & \text{by Lemma A.4} \end{array}$$

Note that the conditions for $\overset{\rightarrow}{\sim}$ are met for all elements in $\mathcal{A}$ and $\mathcal{C}$, and also for M'. Letting $Q' = P_1'$, we have $P' \to Q'$ with $Q \overset{\rightarrow}{\sim} Q'$, as required.

(b) The implication from the previous case does not hold: that is, there is an E' such that

$$P_1' \twoheadrightarrow E'[(\lambda f.M')(\lambda x.O)]$$

while there is no value V' such that $E'[(\lambda f.M')(\lambda x.O)] \twoheadrightarrow E'[V']$. In this case, we choose as our Q' with $P' \xrightarrow{+} Q'$ not P_1' itself, but the expression $E'[M'[f \mapsto \lambda x.O]]$ that P_1' reduces to by virtue of:

$$\begin{array}{rcl} P_1' & \twoheadrightarrow & E'[(\lambda f.M')(\lambda x.O)] \\ & \to & E'[M'[f \mapsto \lambda x.O]] \\ & =: & Q' \end{array}$$

Rather than letting P' simulate P step by step, we have, in a sense, jumped ahead in the simulation. We need to show that we actually stay in the simulation relation with this choice of Q', that is, $Q \sim^{\mathcal{A}}_{\mathcal{C}} Q'$. First, by definition $E_0 \sim^{\mathcal{A}}_{\mathcal{C}\cup\{E'\}} E'$ and $\lambda x.O \approx^{\mathcal{A}}_{\mathcal{C}\cup\{E\}} \lambda x.O$. Therefore:

$$\begin{array}{rcll}
M & \approx^{\mathcal{A}}_{\mathcal{C}} & M' & \\
M & \approx^{\mathcal{A}}_{\mathcal{C}\cup\{E'\}} & M' & \text{by Lemma A.6} \\
M[f \mapsto \lambda x.O] & \approx^{\mathcal{A}}_{\mathcal{C}\cup\{E'\}} & M'[f \mapsto \lambda x.O] & \text{by Lemma A.5} \\
E_0[M[f \mapsto \lambda x.O]] & \sim^{\mathcal{A}}_{\mathcal{C}\cup\{E'\}} & E'[M'[f \mapsto \lambda x.O]] & \text{by Definition of } \sim^{\mathcal{A}}_{\mathcal{C}\cup\{E'\}}
\end{array}$$

Because we assumed it is not the case that $E'[(\lambda f.M')(\lambda x.O)] \twoheadrightarrow E'[V']$ for any V', the condition for $\overset{\rightarrow}{\sim}$ is met for E' as well as for the elements of $\mathcal{A}$ and $\mathcal{C}$, so that we have

$$Q = E_0[M[f \mapsto \lambda x.O]] \overset{\rightarrow}{\sim} E'[M'[f \mapsto \lambda x.O]] =: Q'$$

and we are done, as $P \to Q$, $P' \xrightarrow{+} Q'$, and $Q \overset{\rightarrow}{\sim} Q'$.

4. Case $P = E_0[(\lambda x.O)V]$ and $P' = E'_0[(\lambda y.((\lambda f.M')(\lambda x.O); O))V']$. This can only be the case if $M' \in \mathcal{A}$. Now $(\lambda f.M')(\lambda x.O)$ will be called next:

   $$E'_0[(\lambda y.((\lambda f.M')(\lambda x.O); 0))V'] \to E'_0[(\lambda f.M')(\lambda x.O); O]$$

   By the definition of $\overset{\rightarrow}{\sim}$ and the fact that $M' \in \mathcal{A}$, this call must return to its calling context $E'_0[[\,]; O]$:

   $$E'_0[(\lambda f.M')(\lambda x.O); O] \twoheadrightarrow E'_0[V_2; O] \to E'_0[O]$$

   So $P \to E_0[O]$ and $P' \xrightarrow{+} E'_0[O]$ with $E_0[O] \overset{\rightarrow}{\sim} E'_0[O]$.

5. Case $P = E_0[V]$ and $P' = E'[V']$. where $E' \in \mathcal{C}$. But by the definition of $\overset{\rightarrow}{\sim}$, this is impossible, since E' is invoked with a value V', that is, $P' \twoheadrightarrow E[V']$. So by contradiction, we are done with this case.

□

Programs related by $\overset{\rightarrow}{\sim}$ have essentially the same reduction behaviour, even though the right-hand side one may perform (finitely many) more computational steps.

The simulation relation would not be much use if it related different observables; but is does not:

Lemma 4.5 Let P and P' be expressions in λ_V+**cont** such that $P \overset{\rightarrow}{\sim} P'$. Then P is a numeral iff P' is the same numeral.

Starting from programs related by the simulation relation, we always obtain the same observations.

Lemma 4.6 Let P and P' be closed expressions in λ_V+**cont** with $P \overset{\rightarrow}{\sim} P'$. If $P \xrightarrow{n} Q$, then $P' \xrightarrow{m} Q'$ for some $m \geq n$ and $Q \overset{\rightarrow}{\sim} Q'$. Furthermore, if $P \uparrow$, then $P' \uparrow$.

Proof By induction on the length of the reduction n, and Lemma 4.4. □

The expressions that we want to show contextually equivalent are actually related by the simulation relation whenever they are plugged into the same context:

Lemma 4.7 For any closing context C in λ_V+**cont**, we have

$$C[(\lambda f.M)(\lambda x.O)] \overset{\rightarrow}{\sim} C[(\lambda f.M)(\lambda y.((\lambda f.M)(\lambda x.O); O))]$$

Proof By induction on M one shows that $M \approx^{\emptyset}_{\emptyset} M$. Then also

$$(\lambda f.M)(\lambda x.O) \approx^{\emptyset}_{\emptyset} (\lambda f.M)(\lambda y.((\lambda f.M)(\lambda x.O); O))$$

By induction on C one shows that $P \approx^{\emptyset}_{\emptyset} P'$ implies $C[P] \approx^{\emptyset}_{\emptyset} C[P']$. Therefore

$$C[(\lambda f.M)(\lambda x.O)] \approx^{\emptyset}_{\emptyset} C[(\lambda f.M)(\lambda y.((\lambda f.M)(\lambda x.O); O))]$$

The statement then follows, as the conditions for $\overset{\twoheadrightarrow}{\sim}$ are trivially true for the members of the empty set. $\square$

From the above lemma, we conclude:

Theorem 4.8 (Equivalence) In the language with continuations, λ_V+**cont**, the expressions

$$(\lambda f.M)(\lambda x.O) \qquad \text{and} \qquad (\lambda f.M)(\lambda y.((\lambda f.M)(\lambda x.O); O))$$

are contextually equivalent.

Proof Let

$$\begin{aligned} P_0 &= C[(\lambda f.M)(\lambda x.O)] \\ P_0' &= C[(\lambda f.M)(\lambda y.((\lambda f.M)(\lambda x.O); O))] \end{aligned}$$

By Lemma 4.7, $P_0 \overset{\twoheadrightarrow}{\sim} P_0'$.

Let $P \downharpoonleft n$ for some numeral n. Then by Lemma 4.6, $P' \downharpoonleft Q'$ where $n \overset{\twoheadrightarrow}{\sim} Q'$, hence $Q' = n$, so $P' \downharpoonleft n$ are required.

Conversely, let $P' \downharpoonleft n$. Suppose $P \uparrow$: but then by Lemma 4.6, $P \uparrow$, contradiction. So $P \downharpoonleft Q$. Hence $n = Q$, so $P \downharpoonleft n$ as required. $\square$

Lemma 4.9 Exceptions can separate $(\lambda f.M)(\lambda x.O)$ and $(\lambda f.M)(\lambda y.((\lambda f.M)(\lambda x.O); O))$.

Proof Let M and C be defined as follows.

$$\begin{aligned} M &= (\texttt{handle}\ e\ (\lambda z.\texttt{raise}\ e\ 1)\ (f\ 0); \texttt{raise}\ e\ 0) \\ C &= \texttt{handle}\ e\ (\lambda x.x)\ [\,] \end{aligned}$$

Then we have the following reductions:

$$\begin{aligned} & C[(\lambda f.M)\,(\lambda x.O)] \\ = \ & \texttt{handle}\ e\ (\lambda x.x)\ (\lambda f.(\texttt{handle}\ e\ (\lambda z.\texttt{raise}\ e\ 1)\ (f\ 0); \texttt{raise}\ e\ 0))(\lambda x.O) \\ \twoheadrightarrow\ & 0 \end{aligned}$$

$$\begin{aligned} & C[(\lambda f.M)(\lambda y.((\lambda f.M)(\lambda x.O); O))] \\ = \ & \texttt{handle}\ e\ (\lambda x.x) \\ & \quad (\lambda f.(\texttt{handle}\ e\ (\lambda z.\texttt{raise}\ e\ 1)\ (f\ 0); \texttt{raise}\ e\ 0)) \\ & \quad (\lambda y.((\lambda f.(\texttt{handle}\ e\ (\lambda z.\texttt{raise}\ e\ 1)\ (f\ 0); \texttt{raise}\ e\ 0))(\lambda x.O); O)) \\ \twoheadrightarrow\ & 1 \end{aligned}$$

□

Instead of the formulation with an arbitrary expression M, we could also phrase the equivalence in terms of closed expressions. The two formulations are interderivable if we assume that the β-value law preserves contextual equivalence, which is a standard result.

Corollary 4.10 The following two pure, closed, simply typed λ-terms are contextually equivalent is λ_V+**cont**.

$$\lambda f.f(\lambda x.O) \cong \lambda f.f(\lambda y.(f((\lambda x.O);O)))$$

where O is an arbitrary closed term, such as $\lambda x.x$.

Corollary 4.11 Exceptions cannot macro-express continuations.

Remark 4.12 It may be worth seeing why the equivalence proof for continuations fails for exceptions. What is not true for a language with exceptions is that a continuation is either invoked or ignored. (Or, put differently, the business end of an evaluation context is not just the hole, but also the handlers enclosing the hole.) With exceptions, one can jump to a handler further up inside a context. For the simulation relation $\overset{\twoheadrightarrow}{\sim}$, this means that there can be $P \overset{\twoheadrightarrow}{\sim} P'$ with $P \to Q$ and $P' \to Q'$ such that $Q \overset{\twoheadrightarrow}{\sim} Q'$ does not hold. As an example, consider the following. For $i = 1, 2$, let $E_i = \texttt{handle}\, e\, ((\lambda x.0)\,[\ ])\,(\lambda y.i)$. Then $E_1[\texttt{raise}\, e\, 0] \overset{\twoheadrightarrow}{\sim} E_2[\texttt{raise}\, e\, 0]$, because $E_1[\texttt{raise}\, e\, 0] \sim^{\emptyset}_{\{E_2\}} E_2[\texttt{raise}\, e\, 0]$ and $E_2[\texttt{raise}\, e\, 0] \overset{*}{\not\to} E_2[V]$ for any value V. But $E_i[\texttt{raise}\, e\, 0] \to (\lambda x.i)0$ and $(\lambda x.1)0 \not\overset{\twoheadrightarrow}{\sim} (\lambda x.2)0$.

If we want to relate expressions in the presence of exceptions, we need a simulation relation that is more discriminating, in accounting for handlers. We will see such a relation in the next Section.

5 Exceptions and continuations cannot express state

The aim of this section is to show that the contextual equivalence $M \cong (M; M)$ holds even if exceptions and continuations are present in the language. This is in a sense the basic equivalence for control constructs, variations of which could be broken by one, but not the other control construct. Since assignment easily breaks the equivalence, we draw a line between control and state. This turns out to be a fine line, for if a different way to combine exceptions and continuations in the same language is used, then their combination can in fact express assignment. Furthermore, the proof of this equivalence again uses the technique of arguing, by excluded middle, whether some continuation will or will not be invoked in the future.

Recall that λ_V+**cont**+**state** is the language with both exceptions and continuations; its operational semantics is given in Figure 9.

As in the previous section, we start by defining suitable pre-simulation relations.

Definition 5.1 (Pre-simulation relations) Let $\mathcal{C}$ be a set of closed evaluation contexts. We define relations $\sim_{\mathcal{C}}$ and $\approx_{\mathcal{C}}$, both on expressions and evaluation contexts, by mutual recursion as given in Figure 10, and analogously for evaluation contexts, as given in Figure 11.

As the next lemma shows, pre-simulation $\sim_{\mathcal{C}}$ ensures that related programs behave the same, until either M is found in the evaluation position on the left-hand side and $M'; M'$ on the right; or a continuation from $\mathcal{C}$ is invoked.

Figure 9: Semantics of λ_V+**cont**+**exn**

$$\begin{array}{rcl}
V & ::= & x \mid n \mid \lambda x.M \mid \#E \mid e \\
E & ::= & [\,] \mid (E\ M) \mid (V\ E) \mid (\texttt{succ}\ E) \mid (\texttt{pred}\ E) \mid (\texttt{if0}\ E\ \texttt{then}\ M\ \texttt{else}\ M) \\
 & \mid & (\texttt{throw}\ E\ M) \mid (\texttt{throw}\ V\ E) \\
 & \mid & (\texttt{raise}\ E\ M) \mid (\texttt{raise}\ V\ E) \mid (\texttt{handle}\ e\ E\ N) \mid (\texttt{handle}\ e\ V\ E)
\end{array}$$

$$\begin{array}{lcll}
E[(\lambda x.\ P)\ V] & \to & E[P[x \mapsto V]] & \\
E[\texttt{succ}\ n] & \to & E[n+1] & \\
E[\texttt{pred}\ 0] & \to & E[0] & \\
E[\texttt{pred}\ (n+1)] & \to & E[n] & \\
E[\texttt{if0}\ 0\ \texttt{then}\ M\ \texttt{else}\ N] & \to & E[M] & \\
E[\texttt{if0}\ (n+1)\ \texttt{then}\ M\ \texttt{else}\ N] & \to & E[N] & \\
E[\texttt{rec}\ f(x).\ M] & \to & E[\lambda x.M[f \mapsto \texttt{rec}\ f(x).\ M]] & \\
E[\texttt{callcc}\ M] & \to & E[M(\#E)] & \\
E[\texttt{throw}\ (\#E')\ V] & \to & E'[V] & \\
E[\texttt{handle}\ e\ V_h\ E_e[\texttt{raise}\ e\ V]] & \to & E[V_h V] & E_e \neq E_1[\texttt{handle}\ e\ V_{h2}\ E_2] \\
E[\texttt{handle}\ e\ V_h\ V] & \to & E[V] &
\end{array}$$

Figure 10: Pre-simulation relations on expressions

$$\frac{M \approx_{\mathcal{C}} M' \qquad N \approx_{\mathcal{C}} N'}{MN \approx_{\mathcal{C}} M'N'} \qquad \frac{M \approx_{\mathcal{C}} M' \qquad N \approx_{\mathcal{C}} N'}{\texttt{throw}\ M\ N \approx_{\mathcal{C}} \texttt{throw}\ M'\ N'}$$

$$\frac{M \approx_{\mathcal{C}} M' \qquad N \approx_{\mathcal{C}} N' \qquad L \approx_{\mathcal{C}} L'}{\texttt{if0}\ M\ \texttt{then}\ N\ \texttt{else}\ L \approx_{\mathcal{C}} \texttt{if0}\ M'\ \texttt{then}\ N'\ \texttt{else}\ L'} \qquad \frac{}{n \approx_{\mathcal{C}} n}$$

$$\frac{M \approx_{\mathcal{C}} M'}{\lambda x.M \approx_{\mathcal{C}} \lambda x.M'} \qquad \frac{M \approx_{\mathcal{C}} M'}{\texttt{callcc}\ M \approx_{\mathcal{C}} \texttt{callcc}\ M'}$$

$$\frac{M \approx_{\mathcal{C}} M'}{\texttt{rec}\ f(x).\ M \approx_{\mathcal{C}} \texttt{rec}\ f(x).\ M'} \qquad \frac{}{x \approx_{\mathcal{C}} x}$$

$$\frac{M \approx_{\mathcal{C}} M'}{\texttt{pred}\ M \approx_{\mathcal{C}} \texttt{pred}\ M'} \qquad \frac{M \approx_{\mathcal{C}} M'}{\texttt{succ}\ M \approx_{\mathcal{C}} \texttt{succ}\ M'}$$

$$\frac{M \approx_{\mathcal{C}} M'}{\texttt{raise}\ e\ M \approx_{\mathcal{C}} \texttt{raise}\ e\ M'} \qquad \frac{N \approx_{\mathcal{C}} N' \qquad L \approx_{\mathcal{C}} L'}{\texttt{handle}\ e\ N\ L \approx_{\mathcal{C}} \texttt{handle}\ e\ N'\ L'}$$

$$\frac{}{e \approx_{\mathcal{C}} e}$$

$$\frac{M \approx_{\mathcal{C}} M'}{M \approx_{\mathcal{C}} (M'; M')} \qquad \frac{E \sim_{\mathcal{C}} E'}{\#E \approx_{\mathcal{C}} \#E'}$$

$$\frac{M \approx_{\mathcal{C}} M'}{M \sim_{\mathcal{C}} M'} \qquad \frac{E \sim_{\mathcal{C}} E' \qquad N \approx_{\mathcal{C}} N' \qquad E'[[\,]; M'] \in \mathcal{C}}{E[N] \sim_{\mathcal{C}} E'[N'; M']}$$

Figure 11: Pre-simulation relations on evaluation contexts

$$\frac{E \approx_{\mathcal{C}} E' \qquad N \approx_{\mathcal{C}} N'}{EN \approx_{\mathcal{C}} E'N'} \qquad \frac{V \approx_{\mathcal{C}} V' \qquad E \approx_{\mathcal{C}} E'}{VE \approx_{\mathcal{C}} V'E'}$$

$$\frac{E \approx_{\mathcal{C}} E' \qquad N \approx_{\mathcal{C}} N'}{\texttt{throw}\ E\ N \approx_{\mathcal{C}} \texttt{throw}\ E'\ N'} \qquad \frac{V \approx_{\mathcal{C}} V' \qquad E \approx_{\mathcal{C}} E'}{\texttt{throw}\ V\ E \approx_{\mathcal{C}} \texttt{throw}\ V'\ E'}$$

$$\frac{E \approx_{\mathcal{C}} E' \qquad N \approx_{\mathcal{C}} N' \qquad M \approx_{\mathcal{C}} M'}{\texttt{if0}\ E\ \texttt{then}\ N\ \texttt{else}\ M \approx_{\mathcal{C}} \texttt{if0}\ E'\ \texttt{then}\ N'\ \texttt{else}\ M'} \qquad \frac{}{[\,] \approx_{\mathcal{C}} [\,]}$$

$$\frac{E \approx_{\mathcal{C}} E'}{\texttt{pred}\ E \approx_{\mathcal{C}} \texttt{pred}\ E'} \qquad \frac{E \approx_{\mathcal{C}} E'}{\texttt{succ}\ E \approx_{\mathcal{C}} \texttt{succ}\ E'}$$

$$\frac{V \approx_{\mathcal{C}} V' \qquad E \approx_{\mathcal{C}} E'}{\texttt{raise}\ V\ E \approx_{\mathcal{C}} \texttt{raise}\ V\ E'} \qquad \frac{E \approx_{\mathcal{C}} E' \qquad M \approx_{\mathcal{C}} M'}{\texttt{raise}\ E\ M \approx_{\mathcal{C}} \texttt{raise}\ E'\ M'}$$

$$\frac{E \approx_{\mathcal{C}} E' \qquad M \approx_{\mathcal{C}} M'}{\texttt{handle}\ e\ E\ M \approx_{\mathcal{C}} \texttt{handle}\ e\ E'\ M'} \qquad \frac{V \approx_{\mathcal{C}} V' \qquad E \approx_{\mathcal{C}} E'}{\texttt{handle}\ e\ V\ E \approx_{\mathcal{C}} \texttt{handle}\ e\ V'\ E'}$$

$$\frac{E \approx_{\mathcal{C}} E'}{E \sim^{\mathcal{A}}_{\mathcal{C}} E'} \qquad \frac{E_1 \sim_{\mathcal{C}} E'_1 \qquad E \approx_{\mathcal{C}} E' \qquad E'_1[[\,]; M'] \in \mathcal{C}}{E_1[E] \sim^{\mathcal{A}}_{\mathcal{C}} E'_1[E'; M']}$$

Lemma 5.2 (One-step simulation) Let $P \sim_{\mathcal{C}} P'$. Then (at least) one of the following is the case:

1. $P \not\twoheadrightarrow$ and $P' \not\twoheadrightarrow$.
2. $P \to P_1$ and $P' \to P_1$ with $P'_1 \sim_{\mathcal{C}} P'_1$.
3. $P = E[M]$ and $P' = E'[M'; M']$ with $E \sim_{\mathcal{C}} E'$ and $M \approx_{\mathcal{C}} M'$, but $E'[[\,]; M'] \notin \mathcal{C}$.
4. $P' = E'[V']$ for some $E' \in \mathcal{C}$.

We refine our pre-simulation into a simulation by demanding that no continuation in $\mathcal{C}$ be ever invoked. The last case in Lemma 5.2 is thereby ruled out.

Definition 5.3 Let $P \overset{\twoheadrightarrow}{\sim} P'$ iff $P \sim_{\mathcal{C}} P'$ for some $\mathcal{C}$ such that for all for all $E' \in \mathcal{C}$, there is no value V' with $P' \twoheadrightarrow E'[V']$.

When we are faced with a situation of the form $E[M] \sim_{\mathcal{C}} E'[M'; M']$, the expression M' could again be of the form $M' = E''[M''; M'']$. To avoid an infinite regress, we induct on the derivation.

Definition 5.4 Let $\delta(P, P', \mathcal{C})$ be the least number of instances of the rule

$$\frac{M \approx_{\mathcal{C}} M'}{M \approx_{\mathcal{C}} (M'; M')}$$

necessary to infer $P \sim_{\mathcal{C}} P'$.

Lemma 5.5 Assume $E'[[\,]; M'] \notin \mathcal{C}$. Then

$$\delta(E[M], E'[M'; M'], \mathcal{C}) < \delta(E[M], E'[M'], \mathcal{C})$$

and

$$\delta(E[M], E'[M'; M'], \mathcal{C}) < \delta(E[M], E'[M'; M'], \mathcal{C} \cup \{E'[[\,]; M']\})$$

Lemma 5.6 (Simulation) Let P and P' be closed expressions, and $\mathcal{C}$ a set of closed evaluation contexts of λ_V+**cont**+**exn** such that $P \sim_{\mathcal{C}} P'$ and $P' \upharpoonleft \mathcal{C}$. Then there is a P'_1 with $P' \twoheadrightarrow P'_1$ and moreover:

- If $P \not\twoheadrightarrow$, then $P'_1 \not\twoheadrightarrow$.
- If $P \to Q$, then there is a Q' such that $P'_1 \to Q'$ and $Q \stackrel{\to}{\sim} Q'$.

Proof We induct on $\delta(P, P', \mathcal{C})$. By Lemma 5.2, we need to consider these cases:

1. $P \not\twoheadrightarrow$ and $P' \not\twoheadrightarrow$. Immediate, with $P'_1 = P'$.
2. $P \to Q$ and $P' \to Q'$ with $Q \sim_{\mathcal{C}} Q'$. Then let $P'_1 = P'$. $P' \upharpoonleft \mathcal{C}$ implies $Q' \upharpoonleft \mathcal{C}$, so that $Q \stackrel{\to}{\sim} Q'$.
3. $P = E[M]$ and $P' = E'[M'; M']$ with $E \sim_{\mathcal{C}} E'$ and $M \approx_{\mathcal{C}} M'$. There are two cases depending on whether M' returns a value to its context $E'[[\,]; M']$, or not.

   (a) $E'[M'; M'] \twoheadrightarrow E'[V'; M']$ for some value V'. Hence

   $$E'[M'; M'] \twoheadrightarrow E'[V'; M'] \to E'[M']$$

   Because $E \sim_{\mathcal{C}} E'$ and $M \approx_{\mathcal{C}} M'$, we have $E[M] \sim_{\mathcal{C}} E'[M']$. As $P' \upharpoonleft C$ and $P' \twoheadrightarrow E'[M']$, we have $E'[M'] \upharpoonleft \mathcal{C}$. By Lemma 5.5,

   $$\delta(E[M], E'[M'; M'], \mathcal{C}) < \delta(E[M], E'[M'], \mathcal{C})$$

   so that we can apply the induction hypothesis to $E[M]$, $E'[M']$ and $\mathcal{C}$. That gives us a P'_1 with $E'[M'] \twoheadrightarrow P'_1$. And this P'_1 is what we need, as

   $$P' = E'[M'; M'] \twoheadrightarrow E'[M'] \twoheadrightarrow P'_1$$

   (b) It is not the case that $E'[M'; M'] \twoheadrightarrow E'[V'; M']$ for any value V'. Thus $P' \upharpoonleft E'[[\,]; M']$, in addition to $P' \upharpoonleft \mathcal{C}$, which holds by assumption. Let $\mathcal{C}' = \mathcal{C} \cup \{E'[[\,]; M']\}$; clearly $P' \upharpoonleft \mathcal{C}'$. By Lemma A.10, $E \sim_{\mathcal{C}} E'$ implies $E \sim_{\mathcal{C}'} E'$, and $M \approx_{\mathcal{C}} M'$ implies $M \approx_{\mathcal{C}'} M'$. As $E'[[\,]; M'] \in \mathcal{C}'$, we have $E[M] \sim_{\mathcal{C}'} E'[M'; M']$ by definition of $\sim_{\mathcal{C}'}$. By Lemma 5.5, we have

   $$\delta(E[M], E'[M'; M'], \mathcal{C}') < \delta(E[M], E'[M'; M'], \mathcal{C})$$

   so that we can apply the induction hypothesis to $E[M]$, $E'[M'; M']$ and $\mathcal{C}'$. That gives us a P'_1 as required.
4. $P' = E'[V']$ for some $E' \in \mathcal{C}$. But that is impossible because $P' \twoheadrightarrow E'[V']$ contradicts $P \stackrel{\to}{\sim} P'$.

□

The relation $\stackrel{\to}{\sim}$ does not relate observables only to observables: for instance $n \stackrel{\to}{\sim} (n; n)$, where the right-hand side is not a value. But that is only because $(n; n)$ still needs to do a reduction step to catch up with the left-hand side and become n. In general, observables are only related to (the same) observables if the right-hand side cannot do any more reductions:

Lemma 5.7 Let P and P' be expressions in λ_V+**cont**+**exn** such that $P \stackrel{\to}{\sim} P'$ and $P' \not\twoheadrightarrow$. Then P is a numeral iff P' is the same numeral.

Hence we have the same observations for programs related by $\overset{\rightrightarrows}{\sim}$.

Lemma 5.8 Let P and P' be closed expressions in λ_V+**cont** with $P \overset{\rightrightarrows}{\sim} P'$. If $P \xrightarrow{n} Q$, then $P' \xrightarrow{m} Q'$ for some $m \geq n$ and $Q \overset{\rightrightarrows}{\sim} Q'$. Furthermore, if $P\uparrow$, then $P'\uparrow$.

Lemma 5.9 Let P and P' be expressions in λ_V+**cont**+**exn** such that $P \overset{\rightrightarrows}{\sim} P'$. Then $P \twoheadrightarrow n$ iff $P' \twoheadrightarrow n$.

When we plug M and $M; M$ into the same context, they are in relation $\overset{\rightrightarrows}{\sim}$.

Lemma 5.10 Let M be an expression and C a context in λ_V+**cont**+**exn** such that $C[M]$ is closed. Then $C[M] \overset{\rightrightarrows}{\sim} C[M; M]$.

Proof By induction on M we have $M \approx_\emptyset M$. Hence $M \approx_\emptyset (M; M)$. By induction on C, this implies that $C[M] \approx_\emptyset C[M; M]$, hence also $C[M] \sim_\emptyset C[M; M]$. The condition for $\overset{\rightrightarrows}{\sim}$ is vacuously true for all members of the empty set, so we have $C[M] \overset{\rightrightarrows}{\sim} C[M; M]$. □

From the above lemmas, we can now conclude the desired equivalence.

Theorem 5.11 (Equivalence) In λ_V+**cont**+**exn**, the language with both exceptions and continuations, the expressions M and $(M; M)$ are contextually equivalent.

It is obvious that the putative equivalence $(M; M) \cong M$ is easily broken by assignment: let $M = (x := x + 1; x)$. It is equally clear that exceptions and continuations can break $M; \Omega \cong N; \Omega$; see also [4]. (Here Ω is a divergent expression, such as $(\texttt{rec}\ f(x).\ fx)\,0$ in the typed setting or just $(\lambda x.xx)(\lambda x.xx)$ in the untyped.) In that sense, the equivalence $M \cong (M; M)$ drives a wedge between control and state.

Corollary 5.12 Let O be any pure, closed, simply typed λ-term (such as $\lambda x.x$). The following two pure, closed, simply typed λ-terms are contextually equivalent is λ_V+**cont**+**exn**.

$$\lambda f.f(f\,O; f\,O) \cong \lambda f.f\,O$$

Corollary 5.13 Continuations and exceptions cannot express state.

Proof By Theorem 5.11 and Felleisen [4]. □

Remark 5.14 It is not entirely obvious that continuations and exceptions together cannot give rise to state. In fact, they *almost* can. More precisely, there is a variant of `callcc` and `throw`, called `capture` and `escape` in Standard ML of New Jersey, which can express assignment when it is combined with exceptions: see Figure 12. This allows M and $(M; M)$ to be separated by the choice $M =$ `setx(getx () +1)`. If we use `callcc` instead of `capture` and `throw` instead of `escape`, the same function only gives rise to an uncaught exception, not assignment.

We do not go into any details of the semantics of `capture` and `escape` here, but note in passing that they, unlike the proper continuation constructs, use the handler in an essentially stateful way; that is why the state can be teased out again, as in Figure 12.

Figure 12: State from exceptions and `capture`/`escape`

```
open SMLofNJ.Cont;

exception x of int control_cont;

fun setx n = capture(fn c => (escape c ()) handle x k => escape k n);

setx : int -> unit;

fun getx () =
    let
        val y = capture(fn c2 => raise x c2)
    in
        setx y ; y
    end;

getx: unit -> int;


(setx 0; setx(getx () +1); getx ());
val it = 1 : int
(setx 0; setx(getx () +1); setx(getx () +1); getx ());
val it = 2 : int
```

6 Exceptions and state cannot express continuations

In Section 3, we have contrasted exceptions and continuations in a stateless setting: because continuations allow us to backtrack, we could break an equivalence that holds in the presence of exceptions. The equivalence that we used there would not help us much in the presence of state, since state would already break the equivalence all by itself. On the other hand, we can use some local state to observe backtracking much more easily than we could in the stateless setting.

To observe backtracking, we define an expression that contains a hidden value, so to speak, in that this value cannot be observed from the outside unless a sequence of assignments is run twice. The context is allowed to pass in a non-local function. This function cannot directly glean the values of the local variables. All it could do is use some control behaviour to influence how often the assignments are executed. With exceptions, it can only prevent them from being executed at all by not returning to the call site.

Definition 6.1 We define terms R_1 and R_2 in λ_V+**state** by

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

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.

We need some lemmas that allow us to infer the behaviour of $E[M]$ from that of M.

Lemma 6.2 Let P be an expression, E an evaluation context, and s a store of λ_V+**exn**+**state**. If $s_0, P \twoheadrightarrow s_1, Q$, then $s_0, E[P] \twoheadrightarrow s_1, E[Q]$ by a smaller reduction.

Lemma 6.3 Let M be an expression, E an evaluation context, and s a store of λ_V+**exn**+**state**. If $s, E[M]\downharpoonleft$, then also $s, M\downharpoonleft$.

Lemma 6.4 Let M be an expression and E an evaluation context of λ_V+**exn**+**state**. Let s be a state. If s, M is stuck, then $s, E[M]$ is stuck.

We can confine our attention to M in $E[M]$. We also need to be able to focus on a subset of the store.

Definition 6.5 A set of addresses A is *closed under references in* s, P if $A \subseteq \mathsf{dom}(s)$ $\mathsf{Addr}(P) \subseteq A$ and for all addresses $a \in A$, $\mathsf{Addr}(s(a)) \subseteq A$.

By chasing references emanating from P through the store s, we can never leave A.

The next lemma states that something in the store that is not reachable from the program cannot be changed by it:

Lemma 6.6 If A is closed under references in s, P and $s, P \twoheadrightarrow s_1, Q$, then for all $a \notin A$, $s(a) = s_1(a)$.

We are now ready to define our simulation relation. This relation is in some way quite straightforward, in that we relate R_j and $R_{j'}$ interspersed in some larger program. What is more complicated here is the need to restrict to a subset of the store. Although the difference between R_j and $R_{j'}$ is not observable, their evaluation leaves different integers behind at inaccessible storage addresses.

Definition 6.7 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 $R_j \sim R_{j'}$ for any integers j and j'.
- $\sim$ is extended to evaluation contexts in the evident way:

$$\frac{E \sim E' \qquad M \sim M'}{EM \sim E'M'}$$

  and analogously for all the rules of evaluation-context formation.
- On stores, let $s \sim_A s'$ iff $A \subseteq \mathsf{dom}(s) = \mathsf{dom}(s')$ and for all $a \in A$, $s(a) \sim s'(a)$.
- For stores together with terms, let $s, M \sim_A s', M'$ iff $s \sim_A s'$ and $M \sim M'$ and A is closed under references in both s, M and s', M'.

Briefly, the idea behind the annotation A on the simulation relation $\sim_A$ is that those addresses a with $a \in A$ are guaranteed to be related by $\sim$, whereas those with $a \notin A$ are guaranteed to remain inaccessible.

Lemma 6.8 If $s, P \sim_A s', P'$ and $A \subseteq A_1$, then $s, P \sim_{A_1} s', P'$.

First we show that related programs behave the same until one of them calls R_j.

Lemma 6.9 (One-step simulation) Let P and P' be closed expressions of λ_V+**exn**+**state** such that $s, P \sim_A s', P'$. Then one of the following is the case:

- Both P and P' are values.
- Both s, P and s', P' are stuck.
- Both P and P' are uncaught exceptions, where $P = E[\texttt{raise}\ e\ V]$ and $P' = E'[\texttt{raise}\ e\ V']$ for some values V and V' with $V \sim_A V'$.
- There are values V and V' and contexts E and E' such that $P = E[R_j V]$ and $P' = E'[R_{j'} V']$ with $E \sim E'$ and $V \sim V'$.
- There are expressions P_1 and P_1' and stores s_1 and s_1' such that

  $$\begin{array}{rcl} s, P & \to & s_1, P_1 \\ s', P' & \to & s_1', P_1' \end{array}$$

  such that there is $A \subseteq A_1$ with $(\mathsf{dom}(s) \setminus A) \subseteq (\mathsf{dom}(s_1) \setminus A_1)$ and $s_1, P_1 \sim_{A_1} s_1', P_1'$.

Building on Lemma 6.9, it remains to show that calls of R_j and $R_{j'}$ cannot lead to any observable difference.

Lemma 6.10 (Simulation) Let P and P' be expressions in λ_V+**exn**+**state**, and s and s' be stores such that $s, P \sim_A s', P'$, and $s, P \downharpoonright s_1, Q$. Then there exist a term Q', a store s_1' and a set of addresses A_1 such that

- $s', P' \downharpoonright s_1', Q'$;
- if Q is a value, then $s_1, Q \sim_{A_1} s_1', Q'$;
- if Q is an uncaught exception $E_e[\texttt{raise}\ e\ V]$, then $Q' = E_e'[\texttt{raise}\ e\ V']$ where $s_1, V \sim_{A_1} s_1', V'$;
- $A \subseteq A_1$ and $(\mathsf{dom}(s) \setminus A) \subseteq (\mathsf{dom}(s_1) \setminus A_1)$;

Proof We induct on the size of the reduction: we assume $s, P \downharpoonright s_1, Q$ and that the lemma is true for all smaller reductions. We proceed with the cases given by Lemma 6.9. Suppose both s, P and s', P' are stuck; or both are values; or both are uncaught exceptions. In those cases, we have $s, P \downharpoonright s, P$ and $s', P' \downharpoonright s', P'$, and we are done. Suppose $s, P \to s_1, P_1$. Then $s_1, P_1 \uparrow$ is impossible, as it would imply $s, P \uparrow$. Hence there is are Q and s_2 such that $s_1, P_1 \downharpoonright s_2, Q$. We apply the induction hypothesis to this. Now suppose $P = E[R_j V]$ and $P' = E'[R_{j'} V']$. First, is straightforward to calculate that $R_j V$ applies V to 0 after allocating some addresses that are not known to V:

$$\begin{array}{cll}
 & s, & E[R_j V] \\
= & s, & E[(\lambda f.((\lambda x.\lambda y.(f\,0;\ x \texttt{:=}\ \texttt{!}y;\ y\texttt{:=}j;\ \texttt{!}x))\,(\texttt{ref}\,0)\,(\texttt{ref}\,0))\,V] \\
\to & s, & E[(\lambda x.\lambda y.(V\,0;\ x \texttt{:=}\ \texttt{!}y;\ y\texttt{:=}j;\ \texttt{!}x))\,(\texttt{ref}\,0)\,(\texttt{ref}\,0)] \\
\to & s + \{a \mapsto 0\}, & E[(\lambda x.\lambda y.(V\,0;\ x \texttt{:=}\ \texttt{!}y;\ y\texttt{:=}j;\ \texttt{!}x))\,(\texttt{ref}\,0)\,a] \\
\to & s + \{a \mapsto 0\}, & E[(\lambda y.(V\,0;\ a \texttt{:=}\ \texttt{!}y;\ y\texttt{:=}j;\ \texttt{!}a))\,(\texttt{ref}\,0)] \\
\to & s + \{a \mapsto 0\} + \{b \mapsto 0\}, & E[(\lambda y.(V\,0;\ a \texttt{:=}\ \texttt{!}y;\ y\texttt{:=}j;\ \texttt{!}a))\,b] \\
\to & s + \{a \mapsto 0\} + \{b \mapsto 0\}, & E[V\,0;\ a \texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a]
\end{array}$$

Here $a, b \notin \mathsf{dom}(s)$. Choosing the same addresses, we have the analogous reduction

$$\begin{array}{lll} & s', & E'[R_{j'}V'] \\ \to & s' + \{a \mapsto 0\} + \{b \mapsto 0\}, & E'[V'\,0;\ a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a] \end{array}$$

By Lemma 6.3, $P\downharpoonright$ implies that there must be some N and a store s_1 such that $s + \{a \mapsto 0\} + \{b \mapsto 0\}, V\,0 \downharpoonright s_1, N$. We need to consider whether N is a value, an uncaught exception, or stuck.

- N is a value V_1, that is
$$s + \{a \mapsto 0\} + \{b \mapsto 0\}, V\,0 \downharpoonright s_1, V_1$$
  We apply the induction hypothesis to $s + \{a \mapsto 0\} + \{b \mapsto 0\}, V\,0$, which gives us s_1', V_1' with $s', V'\,0 \downharpoonright s_1', V_1'$ and $s_1, V \sim_{A_1} s_1', V_1'$. Furthermore, $A \subseteq A_1$ and $(\mathsf{dom}(s + \{a \mapsto 0\} + \{b \mapsto 0\}) \setminus A) \subseteq (\mathsf{dom}(s_1) \setminus A_1)$. As $b \notin A$, the this implies $b \notin A_1$.

  As A is closed under references in $s + \{a \mapsto 0\} + \{b \mapsto 0\}, V\,0$, and $b \notin A$, Lemma 6.6 tell us that $s_1(b) = 0$. Furthermore, by Lemma 6.2 applied to the evaluation context $E[[\,];\ a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a]$ we know that
$$s + \{a \mapsto 0\} + \{b \mapsto 0\}, E[V\,0;\ a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a] \twoheadrightarrow s_1, E[V_1;\ a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a]$$
  Therefore
$$\begin{array}{llll} & s + \{a \mapsto 0\} + \{b \mapsto 0\}, & E[V\,0;\ a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a] & \\ \twoheadrightarrow & s_1, & E[V_1;\ a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a] & \\ \to & s_1, & E[a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a] & \\ \to & s_1, & E[a\texttt{:=}\ 0;\ b\texttt{:=}j;\ \texttt{!}a] & \text{as } s_1(b) = 0 \\ \to & s_1 + \{a \mapsto 0\}, & E[b\texttt{:=}j;\ \texttt{!}a] & \\ \to & s_1 + \{a \mapsto 0\} + \{b \mapsto j\}, & E[\texttt{!}a] & \\ \to & s_1 + \{a \mapsto 0\} + \{b \mapsto j\}, & E[0] & \end{array}$$
  In sum, we have for $s, E[R_jV]$ and analogously for $s', E'[R_{j'}V']$ the following reductions:
$$\begin{array}{rcl} s, E[R_jV] & \twoheadrightarrow & s_1, \{a \mapsto 0\} + \{b \mapsto j\}, E[0] \\ s', E'[R_{j'}V'] & \twoheadrightarrow & s_1' + \{a \mapsto 0\} + \{b \mapsto j'\}, E'[0] \end{array}$$
  Moreover, the right-hand sides are related: $E \sim_A E'$ implies $E \sim_{A_1} E'$ by Lemma 6.8, hence $E[0] \sim_{A_1} E'[0]$, also $\mathsf{Addr}(E[0]) = \mathsf{Addr}(E'[0]) \subseteq A \subseteq A_1$. As $s_1 + \{a \mapsto 0\} + \{b \mapsto j\} \sim_{A_1} s_1' + \{a \mapsto 0\} + \{b \mapsto j'\}$, we have
$$s_1 + \{a \mapsto 0\} + \{b \mapsto j\}, E[0] \quad \sim_{A_1} \quad s_1' + \{a \mapsto 0\} + \{b \mapsto j'\}, E'[0]$$
  This is the linchpin of the whole proof: the store address b may hold different integers j and j', respectively; but that is of no consequence, because b, lying outside of A_1, is garbage. Now $s_1 + \{a \mapsto 0\} + \{b \mapsto j\}, E[0] \uparrow$ would again imply $s, P \uparrow$, contrary to our assumption. Hence there are some Q and s_2 such that $s_1 + \{a \mapsto 0\} + \{b \mapsto j\}, E[0] \downharpoonright s_2, Q$ by a reduction smaller than $s, P \downharpoonright s_2, Q$. We apply the induction hypothesis.

- If N is an uncaught exception:
$$s + \{a \mapsto 0\} + \{b \mapsto 0\}, V\,0 \downharpoonright s_1, E_e[\texttt{raise}\ e\ V_1]$$

By Lemma 6.2,

$$\begin{array}{lll} & s + \{a \mapsto 0\} + \{b \mapsto 0\}, & E[V\,0;\ a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a] \\ \twoheadrightarrow & s_1, & E[E_e[\texttt{raise}\ e\ V_1];\ a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a] \end{array}$$

then there are two cases, depending on whether E catches the exception e or not. If it is not caught, we apply the induction hypothesis and are done.

Otherwise, we have $E = E_1[\texttt{handle}\ e\ V_2\ E_2]$. Hence

$$\begin{array}{lll} & s + \{a \mapsto 0\} + \{b \mapsto 0\}, & E_1[\texttt{handle}\ e\ V_2\ E_2[V\,0;\ a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a]] \\ \twoheadrightarrow & s_1, & E_1[\texttt{handle}\ e\ V_2\ E_{e2}[E_e[\texttt{raise}\ e\ V_1;\ a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a]] \\ \rightarrow & s_1, & E_1[V_2 V_1] \end{array}$$

Hence the assignments are discarded; a and b are garbage, as in the previous case, but now they do not even hold different values.

- If s_1, N is stuck, we have by Lemma 6.2 that

$$s, P \twoheadrightarrow s_1, E[N; a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a]$$

and the latter is stuck by Lemma 6.4. We apply the induction hypothesis to

$$s + \{a \mapsto 0\} + \{b \mapsto 0\}, V\,0 \downharpoonleft s_1, N$$

giving us s'_1, N', which is also stuck. Applying Lemmas 6.2 and 6.4 again, we conclude that s', P' is stuck, and we are done.

□

Lemma 6.11 In λ_V+**exn**+**state**, the following contextual equivalence holds:

$$R_1 \cong R_2$$

where $R_j \equiv \lambda f.((\lambda x.\lambda y.(f\,0;\ x\texttt{:=}\ \texttt{!}y;\ y\texttt{:=}j;\ \texttt{!}x))\,(\texttt{ref}\,0)\,(\texttt{ref}\,0))$ as in Definition 6.1.

Proof 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 $\emptyset, C[R_1] \sim_\emptyset \emptyset, C[R_2]$. The statement follows from Lemma 6.10. □

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 6.12 In λ_V+**cont**+**state**, R_1 and R_2 can be separated: there is a context C such that

$$\emptyset, C[R_1] \twoheadrightarrow s_1, 1$$
$$\emptyset, C[R_2] \twoheadrightarrow s'_1, 2$$

We omit the lengthy reduction here, but see Figure 13 for the separating context written in Standard ML of New Jersey.

Figure 13: 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

7 Conclusions

Exceptions and continuations, as we have seen, are fundamentally different in their semantics, and in the way we can reason about them. Consequently they admit different contextual equivalences, so that macro-inexpressiveness follows as a corollary. But more importantly the difference in reasoning elucidates both constructs.

The same basic technique, starting directly from the operational semantics and establishing a simulation relation, worked in all cases. Significant variations arose because of the effects in the language. Both continuations and assignment allow communication between different parts of the program. In proving contextual equivalence, we have to reason about expressions that are not themselves equivalent in all contexts, but only if we restrict the effects that they may use to communicate with the larger program. Hence our induction hypothesis can not simply assume that they behave the same. Rather, we need to carry extra information in the simulation relation. For continuations, this was a set of continuations guaranteed never to be invoked; for state it was a set of storage addresses guaranteed to be the only ones that the expression could reach.

The contrast between exceptions and first-class continuations is in large part due to the fact that exceptions do not allow backtracking, as becomes particularly clear if we can detect backtracking, or its absence, by local state, as we did in Section 6. The same issue can also be addressed using linear typing, to make explicit that `callcc` allows a continuation to be used multiple times, whereas in the presence of only exceptions, continuations are *linearly used* [2], which prevents backtracking. If one could exploit the linearity in the CPS transforms for exceptions to prove equivalences, then the approach based on linear typing, and the one in this paper, based on equivalences, could be related and combined into a larger picture of control constructs.

Another connection to typing and logic may lie in the distinction between classical and intuitionistic logic, with continuations being perhaps inherently classical and exceptions intuitionistic.

At the level of types, there is a well established connection between the types of control operators for first-class continuations and classical logic [7, 3]. More specifically, Griffin [7] has shown that a term that backtracks witnesses excluded middle $\neg A \vee A$. For exceptions, however, there is not such connection to classical logic; to the extent that one can give a logical interpretation to exceptions, it would arguably be intuitionistic [27], the same as for a language without control.

We can perhaps glimpse a connection between continuations and classical logic not only in typing, but also in reasoning about control. The proofs of Lemmas 3.7 and 6.10 about exceptions are intuitionistic, whereas the main arguments about control behaviour in Lemmas 4.4 and 5.6 are crucially classical in their use of excluded middle: it was argued that some continuation would either be invoked at some unknown point in the future, or not. Arguing thus in terms of excluded middle greatly simplified the proof; for the author it was the breakthrough in proving an otherwise recalcitrant equivalence. A more constructive, intuitionistic argument would presumably involve keeping track of the continuation and backtracking in the proof when it is invoked, at the cost of a prohibitive amount of additional housekeeping in the construction of the simulation relation. In a sense, the use of excluded middle neatly encapsulated the backtracking in the logic, with proof by contradiction as the logical analogue of jumping. Rather than having to backtrack whenever a continuation in $\mathcal{C}$ is invoked, we obtain a contradiction, and are done.

Acknowledgements

This paper would not exist without the earlier joint work with Jon Riecke from which it evolved. The $M \cong M; M$ equivalence for continuations was first mentioned to me as an open problem by Andrzej Filinski in 1996. Thanks to Peter O'Hearn for comments and discussion.

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] Josh Berdine, Peter W. O'Hearn, Uday Reddy, and Hayo Thielecke. Linearly used continuations. In Amr Sabry, editor, *Proceedings of the 3rd ACM SIGPLAN Workshop on Continuations*, Indiana University Technical Report No 545, pages 47–54, 2001.

[3] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Proc. ACM Symp. Principles of Programming Languages*, pages 163–173, January 1991.

[4] Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, volume 17, pages 35–75, 1991.

[5] Matthias Felleisen and Daniel P. Friedman. *The Seasoned Schemer.* MIT Press, 1996.

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification (second edition).* Addison-Wesley, 2000.

[7] Timothy G. Griffin. A formulae-as-types notion of control. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, CA USA, 1990.

[8] 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.

[9] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(3):7–105, 1998.

[10] James Laird. A fully abstract games semantics of local exceptions. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science*, 2001.

[11] Peter J. Landin. A generalization of jumps and labels. Report, UNIVAC Systems Programming Research, August 1965.

[12] Peter J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2), 1998. Reprint of [11].

[13] Xavier Leroy and Pierre Weis. *Manual de reference du language Caml.* InterEditions, Paris, 1998.

[14] Mark Lillibridge. Exceptions are strictly more powerful than Call/CC. Technical Report CMU-CS-95-178, Carnegie Mellon University, July 1995.

[15] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[16] John C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.

[17] Jon G. Riecke and Hayo Thielecke. Typed exceptions and continuations cannot macro-express each other. In Jĭrí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Proceedings 26th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *LNCS*, pages 635–644. Springer Verlag, 1999.

[18] 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.

[19] Guy L. Steele. *Common Lisp: the Language.* Digital Press, 1990.

[20] 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.

[21] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, UK, 1974.

[22] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, April 2000. Reprint of [21].

[23] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1997.

[24] Hayo Thielecke. *Categorical Structure of Continuation Passing Style.* PhD thesis, University of Edinburgh, 1997. Also available as technical report ECS-LFCS-97-376.

[25] Hayo Thielecke. Using a continuation twice and its implications for the expressive power of `call/cc`. *Higher-Order and Symbolic Computation*, 12(1):47–74, 1999.

[26] Hayo Thielecke. On exceptions versus continuations in the presence of state. In Gert Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000*, number 1782 in LNCS, pages 397–411. Springer Verlag, 2000.

[27] Hayo Thielecke. Comparing control constructs by typing double-barrelled CPS transforms. In Amr Sabry, editor, *Proceedings of the 3rd ACM SIGPLAN Workshop on Continuations*, Indiana University Technical Report No 545, pages 17–25, 2001.

A Appendix: proof of some lemmas

A.1 Lemmas from Section 3

Lemma A.1 If $M \approx M'$ and $E \approx E'$, then $E[M] \approx E'[M']$.

Proof Induction on the derivation of $E \approx E'$. □

Lemma A.2 If $M \approx M'$ and $N \approx N'$, then $M[x \mapsto N] \approx M'[x \mapsto N']$.

Proof Induction on the derivation of $M \approx M'$. □

Lemma A.3 (Decomposition) Let P and P' be closed such that $P \approx P'$. Then at least one of the following is the case:

1. P and P' are the same numeral n.
2. P and P' are the same exception e.
3. $P = \lambda x.M$ and $P' = \lambda x.M'$.
4. P and P' are both stuck.
5. $P = E[(\lambda f.(f1; f2))M]$ and $P' = E'[(\lambda fg.(f\,1; g\,2))M'M']$
6. $P = E[(\lambda x.M)V]$ and $P' = E'[(\lambda x.M')V']$.
7. $P = E[\texttt{rec}\ f(x).\ M)]$ and $P' = E'[\texttt{rec}\ f(x).\ M')]$
8. $P = E[\texttt{pred}\ n]$ and $P' = E'[\texttt{pred}\ n]$
9. $P = E[\texttt{succ}\ n]$ and $P' = E'[\texttt{succ}\ n]$
10. $P = E[\texttt{if0}\ V\ \texttt{then}\ M\ \texttt{else}\ N]$ and $P' = E'[\texttt{if0}\ V'\ \texttt{then}\ M'\ \texttt{else}\ N']$
11. $P = E[\texttt{handle}\ e\ E_e[\texttt{raise}\ e\ V]\ V_2]$ and $P' = E'[\texttt{handle}\ e\ E'_e[\texttt{raise}\ e\ V']\ V'_2]$
12. $P = E[\texttt{handle}\ e\ V\ V_2]$ and $P' = E'[\texttt{handle}\ e\ V'\ V'_2]$
13. $P = E_e[\texttt{raise}\ e\ V]$ and $P' = E'_e[\texttt{raise}\ e\ V']$

In all cases, $E \approx E'$, $V \approx V'$, $M \approx M'$, $V_2 \approx V'_2$, $N \approx N'$ and $E_1 \approx E'_1$.

Proof Induction on the derivation of $P \approx P'$. □

A.2 Lemmas from Section 4

We first need some technical lemmas to ensure that these relations respect the plugging of an expression into an evaluation context, substitution, and a kind of weakening.

Lemma A.4 Let $E \approx_{\mathcal{C}}^{\mathcal{A}} E'$, $E_1 \sim_{\mathcal{C}}^{\mathcal{A}} E_1'$, $M \approx_{\mathcal{C}}^{\mathcal{A}} M'$. Then

$$E[M] \approx_{\mathcal{C}}^{\mathcal{A}} E'[M'] \qquad E_1[M] \sim_{\mathcal{C}}^{\mathcal{A}} E_1'[M']$$

Proof Induction on the derivation of $E \approx_{\mathcal{C}}^{\mathcal{A}} E'$ and $E_1 \sim_{\mathcal{C}}^{\mathcal{A}} E_1'$. □

Lemma A.5 Let $N \approx_{\mathcal{C}}^{\mathcal{A}} N'$. If $M \approx_{\mathcal{C}}^{\mathcal{A}} M'$, then $M[x \mapsto N] \approx_{\mathcal{C}}^{\mathcal{A}} M'[x \mapsto N']$.

Proof Induction on the derivation of $M \approx_{\mathcal{C}}^{\mathcal{A}} M'$. As elements of $\mathcal{A}$ and $\mathcal{C}$ are required to be closed, the substitution does not change them. □

Lemma A.6 Let $\mathcal{A} \subseteq \mathcal{A}'$ and $\mathcal{C} \subseteq \mathcal{C}'$. Then $\sim_{\mathcal{C}}^{\mathcal{A}} \subseteq \sim_{\mathcal{C}'}^{\mathcal{A}'}$ and $\approx_{\mathcal{C}}^{\mathcal{A}} \subseteq \approx_{\mathcal{C}'}^{\mathcal{A}'}$, both on expressions and evaluation contexts.

Proof Induction on the derivation of $P \sim_{\mathcal{C}}^{\mathcal{A}} P'$, $Q \approx_{\mathcal{C}}^{\mathcal{A}} Q'$, $E_1 \sim_{\mathcal{C}}^{\mathcal{A}} E_1'$, $E \approx_{\mathcal{C}}^{\mathcal{A}} E'$. □

Lemma A.7 (Decomposition) Let P and P' be closed expressions of λ_V+**cont** such that $P \approx_{\mathcal{C}}^{\mathcal{A}} P'$. Then at least one of the following is the case:

- P and P' are the same numeral.
- $P = \lambda x.M$ and $P = \lambda x.M'$,
- P and P' are both stuck.
- $P = E[(\lambda f.N)(\lambda x.0)]$ and $P' = E'[(\lambda f.N')(\lambda y.((\lambda f.N')(\lambda x.0); 0))]$.
- $P = E[(\lambda x.0)V]$ and $P' = E'[(\lambda y.((\lambda f.M_1')(\lambda x.0); 0))V']$ with $M_1' \in \mathcal{A}$.
- $P = E[(\lambda x.M)\, V]$ and $P' = E'[(\lambda x.M')\, V']$.
- $P = E[\mathtt{callcc}\, M]$ and $P' = E'[\mathtt{callcc}\, M']$.
- $P = E[\mathtt{throw}\ (\#E_1)\ V]$ and $P' = E'[\mathtt{throw}\ (\#E_1')\ V']$
- $P = E[\mathtt{rec}\ f(x).\ M]$ and $P' = E[\mathtt{rec}\ f(x).\ M']$
- $P = E[\mathtt{pred}\, n]$ and $P' = E'[\mathtt{pred}\, n]$
- $P = E[\mathtt{succ}\, n]$ and $P' = E'[\mathtt{succ}\, n]$
- $P = E[\mathtt{if0}\ V\ \mathtt{then}\ M\ \mathtt{else}\ N]$ and $P' = E'[\mathtt{if0}\ V'\ \mathtt{then}\ M'\ \mathtt{else}\ N']$

In all cases, $E \approx_{\mathcal{C}} E'$, $V \approx_{\mathcal{C}} V'$, $M \approx_{\mathcal{C}} M'$, $V_2 \approx_{\mathcal{C}}^{\mathcal{A}} V_2'$, $N \approx_{\mathcal{C}}^{\mathcal{A}} N'$ and $E_1 \sim_{\mathcal{C}}^{\mathcal{A}} E_1'$.

Proof The proof is a lengthy induction on the derivation of $P \approx_{\mathcal{C}}^{\mathcal{A}} P'$. As a sample case, consider

$$\frac{M \approx_{\mathcal{C}}^{\mathcal{A}} M' \qquad N \approx_{\mathcal{C}}^{\mathcal{A}} N'}{MN \approx_{\mathcal{C}}^{\mathcal{A}} M'N'}$$

We assume that the statement holds for M and N, and proceed by cases on M.

1. M is a value. Both expressions are stuck unless M is a λ-abstraction. If M is a λ-abstraction, it could be the case that $M = \lambda x.M_1$ and $M' = \lambda x.M_1'$; or that $M = \lambda x.0$ and $M' = \lambda y.((\lambda f.M_1')(\lambda x.0); 0)$. There are several cases what N could be.
   (a) N is a value. Then N' must also be a value. Hence $P = E[MV]$ and $P' = E'[M'V']$, where $E = E' = [\,]$, $V = N$, and $V' = N'$.
   (b) $N = E_1[Q]$. Then $N' = E_1'[Q']$, where $E_1 \approx_{\mathcal{C}}^{\mathcal{A}} E_1'$ and Q and Q' are as required in the various cases. Hence $P = E[Q]$ and $P' = E'[Q']$, where $E = ME_1$ and $E' = M'E_1$, so that $E \approx_{\mathcal{C}}^{\mathcal{A}} E'$.
   (c) N is stuck. Then N' is also stuck. Thus both P and P' are stuck.
2. $M = E_1[Q]$ and $M' = E_1'[Q']$. Then $P = E[Q]$ and $P' = E'[Q']$, where $E = E_1N$ and $E' = E_1'N'$.
3. M is stuck. Then M' must also be stuck. Thus both P and P' are stuck.

□

We need to relate expressions that may also differ in a continuation at the top (related by $\sim_{\mathcal{C}}^{\mathcal{A}}$). Such terms can make the same evaluation step, as long as they do not hit something from $\mathcal{A}$ or $\mathcal{C}$.

Proof of Lemma 4.2.

Proof Assume $P \sim_{\mathcal{C}}^{\mathcal{A}} P'$. By Definition 4.1, there are two cases why $\sim_{\mathcal{C}}^{\mathcal{A}}$ could hold:

- Case $P \approx_{\mathcal{C}}^{\mathcal{A}} P'$. By Lemma A.7, P and P' are either both values, stuck, or of the form $P = E[Q]$ and $P = E[Q']$ with $E \sim_{\mathcal{C}}^{\mathcal{A}} E'$.
- Case $P = E_1[M]$ and $P' = E_1'[M']$ with $M \approx_{\mathcal{C}}^{\mathcal{A}} M'$ and $E_1' \in \mathcal{C}$. If M is a value, then so is M', and we are done. Otherwise, we apply Lemma A.7 to M and M'. That gives us decompositions $M = E_2[Q]$ and $M' = E_2'[Q']$ with $E_2 \approx_{\mathcal{C}}^{\mathcal{A}} E_2'$. Let $E = E_1[E_2]$ and $E' = E_1'[E_2']$; note that again $E \sim_{\mathcal{C}}^{\mathcal{A}} E'$.

So in both cases, we have a decomposition $P = E[Q]$ and $P' = E'[Q']$, where $E \sim_{\mathcal{C}}^{\mathcal{A}} E'$. We proceed by cases on Q and Q', as given by Lemma A.7.

- Case $Q = (\lambda f.N)(\lambda x.0)$ and $Q' = (\lambda f.N')(\lambda y.((\lambda f.N')(\lambda x.0); 0))$: we are done.
- Case $Q = (\lambda x.0)V$ and $Q' = (\lambda y.((\lambda f.M_1')(\lambda x.0); 0))V'$; again we are done.
- Case $Q = (\lambda x.M)V$ and $Q = (\lambda x.M')V'$ with $M \sim_{\mathcal{C}}^{\mathcal{A}} M'$ and $V \sim_{\mathcal{C}}^{\mathcal{A}} V'$. Then

$$\begin{array}{rll} P = & E[(\lambda x.M)V] & \to E[M[x \mapsto V]] \\ P' = & E'[(\lambda x.M')V'] & \to E'[M'[x \mapsto V']] \end{array}$$

  We stay in $\sim_{\mathcal{C}}^{\mathcal{A}}$ because $M[x \mapsto V] \approx_{\mathcal{C}}^{\mathcal{A}} M'[x \mapsto V']$ by Lemma A.5, hence $E[M[x \mapsto V]] \sim_{\mathcal{C}}^{\mathcal{A}} E'[M'[x \mapsto V']]$.

- Case $Q = \texttt{callcc}\, M$ and $Q' = \texttt{callcc}\, M'$. Hence $P = E[\texttt{callcc}\, M]$ and $P' = E'[\texttt{callcc}\, M']$ with $E \sim_{\mathcal{C}}^{\mathcal{A}} E'$ and $M \approx_{\mathcal{C}}^{\mathcal{A}} M'$. Then

$$\begin{array}{rll} P = & E[\texttt{callcc}\, M] & \to E[M\,(\#E)] =: P_1 \\ P' = & E'[\texttt{callcc}\, M'] & \to E'[M'\,(\#E')] =: P_1' \end{array}$$

  We stay in $\sim_{\mathcal{C}}^{\mathcal{A}}$ because $E \sim_{\mathcal{C}}^{\mathcal{A}} E'$ implies $\#E \approx_{\mathcal{C}}^{\mathcal{A}} \#E'$ by Definition 4.1; hence $M\,(\#E) \approx_{\mathcal{C}}^{\mathcal{A}} M'\,(\#E')$ by congruence, so that $E[M\,(\#E)] \sim_{\mathcal{C}}^{\mathcal{A}} E'[M'\,(\#E')]$ by Lemma A.4.

- Case $Q = \texttt{throw}\,(\#E_2)\ V$. Hence $P = E_1[\texttt{throw}\,(\#E_2)\ V]$ and $P' = E_1'[\texttt{throw}\,(\#E_2')\ V']$ with $\#E_2 \approx_{\mathcal{C}}^{\mathcal{A}} \#E_2'$. Then

$$\begin{array}{rll} P = & E[\texttt{throw}\,(\#E_2)\ V] & \to E_2[V] \\ P' = & E'[\texttt{throw}\,(\#E_2')\ V'] & \to E_2'[V'] \end{array}$$

  We stay in $\sim_{\mathcal{C}}^{\mathcal{A}}$ because $\#E_2 \approx_{\mathcal{C}}^{\mathcal{A}} \#E_2'$ implies $E_2 \sim_{\mathcal{C}}^{\mathcal{A}} E_2'$; and this together with $V \approx_{\mathcal{C}}^{\mathcal{A}} V'$, implies $E_2[V] \sim_{\mathcal{C}}^{\mathcal{A}} E_2'[V']$.

- Case $Q = \texttt{pred}\, n$ and $Q' = \texttt{pred}\, n$. Then

$$\begin{array}{rcl} E[\texttt{pred}\, n] & \to & E[n+1] =: P_1 \\ E'[\texttt{pred}\, n] & \to & E'[n+1] =: P_1' \end{array}$$

  We have $n+1 \approx_{\mathcal{C}}^{\mathcal{A}} n+1$, hence $E[n+1] \sim_{\mathcal{C}}^{\mathcal{A}} E'[n+1]$, so that P_1 and P_1' are still related by $\sim_{\mathcal{C}}^{\mathcal{A}}$, as required.

The remaining cases are analogous. □

A.3 Lemmas from Section 5

Lemma A.8 Let $E \approx_{\mathcal{C}} E'$, $E_1 \sim_{\mathcal{C}} E_1'$, $M \approx_{\mathcal{C}} M'$, and $E_2 \approx_{\mathcal{C}} E_2'$. Then

$$\begin{array}{ll} E[M] \approx_{\mathcal{C}} E'[M'] & E_1[M] \sim_{\mathcal{C}} E_1'[M'] \\ E[E_2] \approx_{\mathcal{C}} E'[E_2'] & E_1[E_2] \sim_{\mathcal{C}} E_1'[E_2'] \end{array}$$

Proof Induction on the derivation of $E \approx_{\mathcal{C}} E'$ and $E_1 \sim_{\mathcal{C}} E_1'$. □

Lemma A.9 Let $N \approx_{\mathcal{C}} N'$. If $M \approx_{\mathcal{C}} M'$, then $M[x \mapsto N] \approx_{\mathcal{C}} M'[x \mapsto N']$.

Proof Induction on the derivation of $M \approx_{\mathcal{C}} M'$. As elements of $\mathcal{C}$ are required to be closed, the substitution does not change them. □

Lemma A.10 Let $\mathcal{C} \subseteq \mathcal{C}'$. Then $\sim_{\mathcal{C}} \subseteq \sim_{\mathcal{C}'}$ and $\approx_{\mathcal{C}} \subseteq \approx_{\mathcal{C}'}$, both on expressions and evaluation contexts.

Lemma A.11 Let E and E' be two evaluation contexts of λ_V+**cont**+**exn** with $E \approx_{\mathcal{C}} E'$ and e an exception. Then either $E = E_e$ and $E' = E_e'$, or

$$\begin{array}{rcl} E & = & E_1[\texttt{handle}\ e\ V\ E_e] \\ E' & = & E_1'[\texttt{handle}\ e\ V'\ E_e'] \end{array}$$

where $E_1 \approx_{\mathcal{C}} E_1'$ and $V \approx_{\mathcal{C}} V'$.

Proof Induction on the derivation of $E \approx_{\mathcal{C}} E'$. □

Lemma A.12 Let E and E' be two evaluation contexts of λ_V+**cont**+**exn** with $E \sim_{\mathcal{C}} E'$ and e an exception. Then either $E = E_e$ and $E' = E_e'$, or

$$\begin{array}{rcl} E & = & E_1[\texttt{handle}\ e\ V\ E_e] \\ E' & = & E_1'[\texttt{handle}\ e\ V'\ E_e'] \end{array}$$

where $E_1 \sim_{\mathcal{C}} E_1'$ and $V \approx_{\mathcal{C}} V'$.

Proof We induct on the derivation of $E \sim_{\mathcal{C}} E'$. If $E \sim_{\mathcal{C}} E'$, then we apply Lemma A.11 and conclude from $E_1 \approx_{\mathcal{C}} E_1'$ that $E_1 \sim_{\mathcal{C}} E_1'$, as required. Otherwise $E = E_1[E_2]$ and $E' = E_1'[E_2'; M']$ with $E_1 \sim_{\mathcal{C}} E_1'$, $E_2 \approx_{\mathcal{C}} E_2$ and $E_1'[[\,]; M'] \in \mathcal{C}$. We proceed by cases, depending on whether E_2 handles e.

- Suppose E_2 handles e. Then

$$\begin{array}{rcl} E_2 & = & E_3[\texttt{handle}\, e\, V\ E_{e3}] \\ E_2' & = & E_3'[\texttt{handle}\, e\, V'\ E_{e3}'] \end{array}$$

where $V \approx_{\mathcal{C}} V'$ and $E_{e3} \approx_{\mathcal{C}} E_{e3}'$. Thus

$$\begin{array}{rcl} E & = & E_1[E_3[\texttt{handle}\, e\, V\ E_{e3}]] \\ E' & = & E_1'[E_3'[\texttt{handle}\, e\, V'\ E_{e3}']; M'] \end{array}$$

and $E_1[E_3] \sim_{\mathcal{C}} E_1'[E_3'; M']$ by the definition of $\sim_{\mathcal{C}}$.

- Suppose E_2 does not handle e. By Lemma A.11, E_2' does not handle e either. There are two subcases, depending on whether E_1 handles e or not. If it does not, then by the induction hypothesis, neither does E_1'. Hence $E_1[E_2]$ and $E_1'[E_2']$ both do not handle e, and we are done. If E_1 handles e, then by the induction hypothesis so does E_1', that is

$$\begin{array}{rcl} E_1 & = & E_3[\texttt{handle}\, e\, V\ E_{e3}]] \\ E_1' & = & E_3'[\texttt{handle}\, e\, V'\ E_{e3}']; M'] \end{array}$$

Thus

$$\begin{array}{rcl} E & = & E_3[\texttt{handle}\, e\, V\ E_{e3}[E_2]] \\ E' & = & E_3'[\texttt{handle}\, e\, V'\ E_{e3}'[E_2']; M'] \end{array}$$

and $E_3 \sim_{\mathcal{C}} E_3'[[\,]; M']$ (because $[\,] \approx_{\mathcal{C}} [\,]$).

□

Lemma A.13 Let P and P' be closed expressions of λ_V+**cont**+**exn** with $P \approx_{\mathcal{C}} P'$. Then at least one of the following is the case:

- P and P' are the same numeral n.
- P and P' are the same exception e.
- $P = \lambda x.M$ and $P' = \lambda x.M'$.
- P and P' are both stuck.
- $P = E[M]$ and $P' = E'[M'; M']$.
- $P = E[(\lambda x.M)V]$ and $P' = E'[(\lambda x.M')V']$.
- $P = E[\texttt{callcc}\, M]$ and $P' = E'[\texttt{callcc}\, M']$.
- $P = E[\texttt{throw}\ (\#E_1)\ V]$ and $P' = E'[\texttt{throw}\ (\#E_1')\ V']$

- $P = E[\texttt{rec}\ f(x).\ M]$ and $P' = E'[\texttt{rec}\ f(x).\ M']$
- $P = E[\texttt{pred}\ n]$ and $P' = E'[\texttt{pred}\ n]$
- $P = E[\texttt{succ}\ n]$ and $P' = E'[\texttt{succ}\ n]$
- $P = E[\texttt{handle}\ e\ V_h\ E_e[\texttt{raise}\ e\ V]]$ and $P' = E'[\texttt{handle}\ e\ V_h\ E'_e[\texttt{raise}\ e\ V']]$
- $P = E[\texttt{handle}\ e\ V\ V_2]$ and $P' = E'[\texttt{handle}\ e\ V'\ V'_2]$
- $P = E[\texttt{if0}\ V\ \texttt{then}\ M\ \texttt{else}\ N]$ and $P' = E'[\texttt{if0}\ V'\ \texttt{then}\ M'\ \texttt{else}\ N']$
- $P = E_e[\texttt{raise}\ e\ V]$ and $P' = E'_e[\texttt{raise}\ e\ V']$

In all cases, $E \approx_{\mathcal{C}} E'$, $V \approx_{\mathcal{C}} V'$, $M \approx_{\mathcal{C}} M'$, $V_2 \approx_{\mathcal{C}} V'_2$, $N \approx_{\mathcal{C}} N'$ and $E_1 \sim_{\mathcal{C}} E'_1$.

Proof By induction on $P \approx_{\mathcal{C}} P'$ and cases of the definition of evaluation context, analogous to Lemma A.7. $\square$

The proof of Lemma 5.2 is largely analogous to that of Lemma 4.2, using Lemmas A.13 and A.11.

A.4 Proofs from Section 6

Lemma A.14 (Decomposition) Let P and P' be expressions in λ_V+**exn**+**state**, and s and s' be stores, such that $s, P \sim_A s', P'$. Then at least one of the following is the case:

1. P and P' are the same numeral n.
2. P and P' are the same exception e.
3. P and P' are the same variable x.
4. $P = \lambda x.M$ and $P' = \lambda x.M'$.
5. P and P' are both stuck.
6. $P = E[R_j V]$ and $P' = E'[R_{j'} V']$ for some integers j and j'.
7. $P = E[\texttt{ref}\ V]$ and $P' = E'[\texttt{ref}\ V']$
8. $P = E[\texttt{!}a]$ and $P' = E'[\texttt{!}a]$ for some address a.
9. $P = E[a\texttt{:=}V]$ and $P' = E'[a\texttt{:=}V']$ for some address a.
10. $P = E[\texttt{pred}\ n]$ and $P' = E'[\texttt{pred}\ n]$
11. $P = E[\texttt{succ}\ n]$ and $P' = E'[\texttt{succ}\ n]$
12. $P = E[\texttt{if0}\ V\ \texttt{then}\ M\ \texttt{else}\ N]$ and $P' = E'[\texttt{if0}\ V'\ \texttt{then}\ M'\ \texttt{else}\ N']$
13. $P = E[\texttt{handle}\ e\ V_h\ E_e[\texttt{raise}\ e\ V]]$ and $P' = E'[\texttt{handle}\ e\ V'_h\ E'{}_e[\texttt{raise}\ e\ V']]$
14. $P = E[\texttt{handle}\ e\ V_h\ V]$ and $P' = E'[\texttt{handle}\ e\ V'_h\ V']$
15. $P = E_e[\texttt{raise}\ e\ V]$ and $P' = E'_e[\texttt{raise}\ e\ V']$

Moreover, in all cases $E \sim_A E'$, $V \sim_A V'$, $V_2 \sim_A V'_2$, $M \sim_A M'$, $N \sim_A N'$.

Next we sketch the proof of Lemma 6.9.

Proof We need to consider the cases given by Lemma A.14. The most important ones are those that manipulate the state.

1. Case $P = E[\texttt{ref}\,V]$ and $P' = E'[\texttt{ref}\,V']$ with $E \sim_A E'$ and $V \sim_A V'$. Then $s, E[\texttt{ref}\,V] \to s + \{a \mapsto V\}, E[V]$, where $a \notin \mathsf{dom}(s)$. Hence $s', E'[\texttt{ref}\,V'] \to s' + \{a \mapsto V'\}, E[V']$. (We can pick the same address a, because $a \notin \mathsf{dom}(s') = \mathsf{dom}(s)$.) Now

   $$s + \{a \mapsto V\}, E[V] \sim_{A\cup\{a\}} s' + \{a \mapsto V'\}, E'[V']$$

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

2. Case $P = E[\texttt{!}a]$ and $P' = E'[\texttt{!}a]$ with $E \sim_A E'$. By the definition of $\sim_A$, we have $s(a) \sim_A s'(a)$. Hence $E[s(a)] \sim_A E'[s'(a)]$ and thus $s, E[s(a)] \sim_A s', E'[s'(a)]$.

3. Case $P = E[a\texttt{:=}V]$ and $P' = E'[a\texttt{:=}V']$ with $E \sim_A E'$ and with $V \sim_A V'$. Since $V \sim_A V'$, we have $s + \{a \mapsto V\} \sim_A s' + \{a \mapsto V'\}$ and $E[V] \sim_A E'[V']$, so that $s + \{a \mapsto V\}, E[V] \sim_A s' + \{a \mapsto V'\}$.

$\square$