Combining and Relating Control Effects and their Semantics

James Laird
Department of Computer Science, University of Bath, UK

Combining local exceptions and first class continuations leads to programs with complex control flow, as well as the possibility of expressing powerful constructs such as resumable exceptions. We describe and compare games models for a programming language which includes these features, as well as higher-order references. They are obtained by contrasting methodologies: by annotating sequences of moves with "control pointers" indicating where exceptions are thrown and caught, and by composing the exceptions and continuations monads.

The former approach allows an explicit representation of control flow in games for exceptions, and hence a straightforward proof of definability (full abstraction) by factorization, as well as offering the possibility of a semantic approach to control flow analysis of exception-handling. However, establishing soundness of such a concrete and complex model is a non-trivial problem. It may be resolved by establishing a correspondence with the monad semantics, based on erasing explicit exception moves and replacing them with control pointers.

1 Introduction

Control effects such as exceptions and continuations are key features of higher-order programming languages. They are typically used to recover from errors, and may result in complicated and unpredictable control flow in programs. Therefore, principles for reasoning about notions such as *exception safety* are potentially useful and important. Denotational semantics provides one basis for such principles. Here, broadly speaking, there are two approaches to describing computational effects. Constructions such as *monads*, and *continuation-passing-style interpretations* yield useful algebraic theories for reasoning *soundly* about programs, although they impose additional layers of definition and interpretation through which reasoning about programs must be filtered, particularly in the presence of properties such as locality. By contrast, *game semantics* provides a framework in which to model combinations of effects more directly by the relaxation of constraints on strategies representing functional programs. This approach has been used successfully to give *fully abstract* interpretations of many features, including an account of locality for features such as state [1]. However, the combinatorial nature of games models means that reasoning about denotations — for example, proving basic soundness results — can be difficult in the absence of structuring principles.

Thus it can be useful to relate the direct (games) and indirect (monads, CPS) approaches to effects, to gain the advantages of both representations. This paper will do so for exceptions and continuations. In the process, we construct a first fully abstract model for a language which combines continuations and locally declared exceptions, as in Standard ML of New Jersey. Although many control structures can be implemented using either feature, exceptions and continuations exhibit several subtle but significant differences in behaviour: one way of understanding these is by studying the interaction of the two effects in combination. (For example, observing that exceptions break key equational rules which hold for continuations [8].) Combining exceptions and continuations also provides a way of interpreting further, powerful control constructs: they may be used to macro-express *resumable* exceptions, and implement dynamic delimited control operators such as *prompts* [3].

U. de'Liguoro and A. Saurin (Eds.):
Control Operators and their Semantics 2013 (COS'13)
EPTCS 127, 2013, pp. 113–129, doi:10.4204/EPTCS.127.8

Exceptions and continuations also provide a test case for semantic theories of combining algebraic effects, studied in detail in [4]. Here we shall simply use the fact that there is a distributive law of the monad $\_ + E$ (exceptions) over R^{R^-} (continuations) (which exist for objects E and R whenever the relevant categorical constructions do), since the exceptions monad distributes over any other monad. Thus we have an exceptions-and-continuations monad $R^{R^{-+E}}$.

How can this monad be related to a game semantic account? In the case of first-class continuations (on their own), there is a simple correspondence between the games and monadic interpretations — relaxing the *well-bracketing condition* on strategies renders the lifted sum monad $\Sigma_\_$ introduced in [2] isomorphic to the continuations monad R^{R^-}, where R is the "one-move game", giving both direct and indirect (continuation-passing style) interpretations of call/cc [6].

The case of exceptions is more complicated. We may interpret a single global exception by adding to our games distinguished "exception answer" moves for each question. Extending the continuations monad with such an answer yields a monad formally equivalent to $R^{R^{-+1}}$[1] In the presence of local state, this is sufficient to macro-express local exception declaration, as we may use imperative variables both as flags to indicate which exception has been set, and to carry exceptional values. However, this leaves open the problem of identifying the elements of the model definable using local exception handling, and their intrinsic equivalence.

Exceptions have also been more difficult to incorporate into the simple picture of relaxing constraints on strategies to get more powerful effects; locally declared exceptions can be interpreted directly by relaxing the bracketing condition to a "weak bracketing" condition [7], but fully capturing this behaviour also requires new information to be added to strategies in the form of additional "control pointers" attached to sequences. Relaxing the weak bracketing condition also gives a straightforward and intuitively natural alternative denotation for call-with-current-continuation in the context of exception handling — playing a control pointer to a "closed" move allows the handler-context to be reset. However, this representation of continuations and exceptions is rather implicit, and does not lend itself to reasoning about equivalence between denotations of programs — even to the limited extent of proving soundness with respect to the operational semantics.

The solution adopted here is a correspondence with the exceptions monad and CPS interpretations, given by relating exception-arenas to control games by replacing exception moves with control pointers (in this case, indicating which question is *pending* when an exception is thrown). Finally, we prove full abstraction results for the control games models using *factorization* into the model with only local control defined in [1].

2 An Effectful Functional Programming Language

We shall first describe a simply-typed call-by-value programming language $\mathscr{L}_{\mathscr{RCE}}$ with (locally declared) general references, first-class continuations and local exceptions (which might be considered as a simply-typed fragment of SML of New Jersey). The core of the language, $\mathscr{L}$, is a simply-typed call-by-value λ-calculus based on the computational λ-calculus [11].

Types of $\mathscr{L}$ are generated from the product, sum (and their units 1 and 0) and function types:
$S,T := 0 \mid 1 \mid S \times T \mid S + T \mid S \to T$
We distinguish *computation* and *value* terms. *Values* are given by the grammar:
$U,V := \quad x \mid () \mid \langle U,V \rangle \mid \mathrm{in}_1(\mathrm{V}) \mid \mathrm{in}_2(\mathrm{V}) \mid \lambda \mathrm{x.M}$

[1] Another approach in [12] also uses an exceptions monad on a category of nominal games — here we focus on more concrete models with implicit state.

$$\frac{}{\Gamma,x{:}T\vdash_v x{:}T} \qquad \frac{\Gamma\vdash_v V{:}T}{\Gamma\vdash_c [V]{:}T} \qquad \frac{\Gamma\vdash_c M{:}S \quad \Gamma,x{:}S\vdash_c N{:}T}{\Gamma\vdash_c \mathtt{let}\, x{=}M\, \mathtt{in}\, N{:}T}$$

$$\frac{}{\Gamma\vdash_v ():1} \qquad \frac{\Gamma\vdash_v V{:}0}{\Gamma\vdash_c \mathtt{void}\, V{:}T}$$

$$\frac{\Gamma\vdash_v U{:}S \quad \Gamma\vdash_v V{:}T}{\Gamma\vdash_v \langle U,V\rangle{:}S\times T} \qquad \frac{\Gamma\vdash_v V{:}S\times S' \quad \Gamma,x{:}S,y{:}S'\vdash_c M{:}T}{\Gamma\vdash_c \mathtt{match}\, V\, \mathtt{as}\, (x,y).M{:}T}$$

$$\frac{\Gamma\vdash_v U{:}S}{\Gamma\vdash_v \mathrm{in_i(V)}{:}\mathrm{T_1{+}T_2}}\, i\in\{1,2\} \qquad \frac{\Gamma\vdash_v V{:}S{+}S' \quad \Gamma,x{:}S\vdash_c M{:}T \quad \Gamma,x{:}S'\vdash_c N{:}T}{\Gamma\vdash_c \mathtt{case}\, V\, \mathtt{as}\, \mathtt{in}_1(x).M|\mathrm{in_2(x).N{:}T}}$$

$$\frac{\Gamma,x{:}S\vdash_c M{:}T}{\Gamma\vdash_v \lambda x.M{:}S\to T} \qquad \frac{\Gamma\vdash_v U{:}S\to T \quad \Gamma\vdash_v V{:}S}{\Gamma\vdash_c U\, V{:}T}$$

Table 1: Typing Judgements for Computations and Values

Computations are given by the grammar:

$M,N :=\quad [V] \mid \mathtt{let}\, x = M\, \mathtt{in}\, N \mid \mathtt{void}\, V \mid U\, V \mid \mathtt{match}\, V\, \mathtt{as}\, (x,y).M \mid \mathtt{case}\, V\, \mathtt{as}\, \mathrm{in_1(x).M|in_2(x).N}$

Typing judgements, of the form $\Gamma \vdash_c M : T$ for computations, and $\Gamma \vdash_v V : T$ for values, are given in Table 1. We write $M;N$ for $\mathtt{let}\, x = M\, \mathtt{in}\, N$, if x is not free in M or N.

2.1 Computational Effects

Computational effects are introduced by adding constructs for declaring references and exceptions, and capturing the current continuation as a first-class function, as follows:

References The type $\mathtt{var}[T]$ of references to values of type T is *defined* to be $(T \to 1) \times (1 \to T)$ — the product of the types of its methods, assignment and dereferencing, which may be recovered by left and right projection, respectively — i.e. given $a : \mathtt{var}[T]$ and $V : T$, we sugar $\mathtt{match}\, a\, \mathtt{as}\, (x,y).x\, V$ as $a := V$, and $\mathtt{match}\, a\, \mathtt{as}\, (x,y).y\, ()$ as $\mathtt{deref}(a)$.

Thus the only further syntax we need to add to our type theory is a constant (value) $\mathtt{new} : 1 \to \mathtt{var}[T]$ for declaring a new reference. We write $\mathtt{let}\, x = (\mathtt{new}\, ())\, \mathtt{in}\, x := V; M$ as $\mathtt{new}\, x := V.M$.

Exceptions The type of $\mathtt{exn}$ of exceptions is similarly defined to be the product $((1 \to 0) \to 1) \times (1 \to 0)$ of its method types: *throwing* of type $1 \to 0$ and *catching*, of type $(1 \to 0) \to 1$.[2] Given $e : \mathtt{exn}$, we sugar $\mathtt{match}\, e\, \mathtt{as}\, (x,y).x\, \lambda().N$ and $\mathtt{match}\, e\, \mathtt{as}(x,y).y\, ()$ as $\mathtt{catch}\, e\, \mathtt{in}\, N$ and $\mathtt{throw}(e)$, respectively.

Thus to extend our type theory with exceptions it is sufficent to add a value $\mathtt{new\_exn} : 1 \to \mathtt{exn}$ for declaring a new exception.

Continuations As in New Jersey SML, we introduce first-class continuations via a value $\mathtt{callcc} : ((T \to S) \to T) \to T$, which passes a first class representation of the current *continuation* (as a value of type $T \to S$ for arbitrary S) to its argument.

[2]The "thunked" empty type $1 \to 0$ is used to represent the type of computations which do not return a value.

$$
\begin{array}{lcl}
E[\texttt{case in}_i(\mathrm{V})\ \texttt{as in}_1(\mathrm{x}).\mathrm{M}_1|\texttt{in}_2(\mathrm{x}).\mathrm{M}_2], \mathscr{E} & \longrightarrow & E[M_i[V/x]], \mathscr{E} \\
E[\texttt{match}\ \langle U, V\rangle\ \texttt{as}\ (x,y).M], \mathscr{E} & \longrightarrow & E[M[U/x, V/y]], \mathscr{E} \\
E[(\lambda x.M)\ V], \mathscr{E} & \longrightarrow & E[M[V/x]], \mathscr{E} \\
E[\texttt{let}\ x = [V]\ \texttt{in}\ M], \mathscr{E} & \longrightarrow & E[M[V/x]], \mathscr{E} \\
E[\texttt{new}\ ()], \mathscr{E}[(\mathsf{loc})] & \longrightarrow & E[[\langle \texttt{set}(a), !a\rangle]], \mathscr{E}[\mathsf{loc} \cup \{a\}] \\
E[\texttt{set}(a)\ V], \mathscr{E}[\mathsf{S}] & \longrightarrow & E[[()]], \mathscr{E}[\mathsf{S}[a \mapsto V]] \\
E[!a\ ()], \mathscr{E}[\mathsf{S}] & \longrightarrow & E[[\mathsf{S}(a)]], \mathscr{E} \\
E[\texttt{new\_exn}\ ()], \mathscr{E}[\mathsf{Ex}] & \longrightarrow & E[[\langle \texttt{catch}(e), \texttt{throw}(e)\rangle]], \mathscr{E}[\mathsf{Ex} \cup \{e\}] \\
E[\texttt{catch}(e)\ \lambda x.E_e[\texttt{throw}(e)\ ()]], \mathscr{E} & \longrightarrow & E[[()]], \mathscr{E} \\
E[\texttt{callcc}\ V], \mathscr{E} & \longrightarrow & E[V\ \lambda x.\#E[x]], \mathscr{E} \\
E[\#(M)], \mathscr{E} & \longrightarrow & M, \mathscr{E}
\end{array}
$$

Table 2: Operational Semantics of $\mathscr{L}_{\mathscr{R}\mathscr{C}\mathscr{E}}$

We make use of the following fragments of $\mathscr{L}_{\mathscr{R}\mathscr{C}\mathscr{E}}$ — the purely functional fragment $\mathscr{L}$, the fragment $\mathscr{L}_{\mathscr{R}}$ with local control (i.e. references but no continuations or exceptions, omitting the constants `new_exn` and `callcc`: this is essentially the language defined in [1], with its games model), and the fragment $\mathscr{L}_{\mathscr{R}\mathscr{C}}$ with continuations and references but no exceptions.

2.2 Operational Semantics

To give an operational semantics for $\mathscr{L}_{\mathscr{R}\mathscr{C}\mathscr{E}}$, we introduce constants representing the capacity to read from and write to a location, and raise and handle an exception, and a new constructor, representing composition with the top-level continuation. Let $\mathscr{L}^{\#}_{\mathscr{C}\mathscr{E}}$ be the extension of $\mathscr{L}_{\mathscr{R}\mathscr{C}\mathscr{E}}$ with:

- An unbounded set of pairs of constants $(\texttt{set}(a), !a)$.
- An unbounded set of pairs of constants $(\texttt{throw}(e), \texttt{catch}(e))$.
- An operation $\#\_$ taking computations of type 1 to computations of type T.

Evaluation contexts $E[\_]$ are given by the grammar:

$$E[\_] ::= [\_] \mid \texttt{let}\ x = E[\_]\ \texttt{in}\ M \mid \texttt{catch}(e)\ \lambda x.E[\_]$$

$E_h[\_]$ denotes an evaluation context without a $\texttt{catch}(h)\ \lambda x.\_$ in the spine — i.e. given by the above grammar subject to $e \neq h$.

The "small-step" operational semantics for reducing a term in an environment $\mathscr{E}$ (a set of location names loc and store S, and a set of exception names Ex) is given in (Table 2). Variable names not occurring on the left of a rule are assumed fresh. For a program (computation) $M : 1$, we write $M \Downarrow$ if $M, \varnothing$ reduces to $[()]$. Observational approximation and equivalence are defined with respect to this notion of convergence: $M \lesssim N$ if for all closing contexts, $C[\_] : 1$, $C[M] \Downarrow$ implies $C[N] \Downarrow$. $M \approx N$ if $M \lesssim N$ and $N \lesssim M$.

2.3 Expressiveness

We make some remarks on the expressiveness of our language. Although we have used a simplified version of exceptions which do not carry explicit values, we may macro-express value-carrying exceptions by using references to pass values through the store. For example, for any type T, define the type $\texttt{exn}[T]$

of exceptions carrying values of type T to be $((1 \to 0) \to T) \times (T \to 0)$, so that applying right-projection to a value raises an exception with that value, and applying left projection to a (thunked) computation captures an exception and returns the value it carries. Then we may define an object declarating an exception of type T — $\mathtt{new\_exn}_T : 1 \to \mathtt{exn}[T] =_{df}$

$$\lambda().\mathtt{new}\, a.\mathtt{new\_exn}\, e.[\langle \lambda f.\mathtt{catch}\, e\, \mathtt{in}\, (f\,());\mathtt{deref}(a), \lambda x.(a := x);\mathtt{throw}(e)\rangle]$$

We may represent ML or Java-style exception *handling* — i.e. including code to be run if only if a given exception is caught — by using exceptions *or* continuations to escape from the handler context if an exception is not raised, defining e.g.

$$\mathtt{handle}\, e\, \mathtt{in}\, N\, \mathtt{with}\, M =_{df} \mathtt{callcc}(\lambda k.(\mathtt{catch}\, e\, \mathtt{in}\, N;(k\,()));M)$$

By combining references, exceptions and continuations we may express *resumable exceptions* which may return to the point at which they were raised. e.g. define the declaration $\mathtt{resumable\_exn} : ((1 \to 0) \to (T \to 0)) \times (1 \to T)$ as follows:

$$\mathtt{new}\, a\, \mathtt{in}\, \mathtt{new\_exn}\, e\, \mathtt{in}\, [\langle \lambda f.(\mathtt{catch}\, e\, \mathtt{in}\, f\,());\mathtt{deref}(a), \mathtt{callcc}(\lambda k.a := k;\mathtt{throw}(e))\rangle]$$

Right projection captures the current continuation and raises a (local) exception, left projection traps the exception and returns the continuation from the point it was thrown as a first-class function.

Finally, we note that exceptions and continuations are used in [3] to implement prompts in Standard ML of New Jersey. Prompts are a form of locally declared, dynamically bound, delimited control operator which may be used to express local exceptions, as defined here, and a *delimited* form of `callcc`. However, the implementation of prompts in $\mathrm{SML_{NJ}}$ uses global variables and is not therefore fully compositional: we leave a semantic investigation of the relationship between exceptions, continuations and delimited control as future work.

3 Denotational Semantics: Preliminaries

First, we fix what we mean by a model of the type-theory $\mathscr{L}$ (essentially, a model of the computational λ-calculus [11]):

- a category $\mathscr{C}$ with finite, distributive coproducts and products (including terminal and initial objects) and
- a strong monad $(\Sigma, \eta, \mu, \tau)$ on $\mathscr{C}$ such that for any A and B in $\mathscr{C}$, the exponential $A \Rightarrow \Sigma B$ exists.

Types are interpreted as objects of $\mathscr{C}$ — $[\![1]\!]$ and $[\![0]\!]$ are the terminal and initial objects and $[\![S \times T]\!] = [\![S]\!] \times [\![T]\!]$, $[\![S + T]\!] = [\![S]\!] + [\![T]\!]$ and $[\![S \to T]\!] = [\![S]\!] \Rightarrow \Sigma[\![T]\!]$.
For a context $\Gamma = x_1 : T_1, \ldots, x_n : T_n$, define $[\![\Gamma]\!] = [\![T_1]\!] \times \ldots \times [\![T_n]\!]$.

- *Values* $\Gamma \vdash_v V : T$ are interpreted as morphisms from $[\![\Gamma]\!]$ to $[\![T]\!]$.
- *Computations* $\Gamma \vdash_c M : T$ are interpreted as morphisms from $[\![\Gamma]\!]$ to $\Sigma[\![T]\!]$.

Formal semantics are given in Table 3.

$$
\begin{array}{lcl}
[\![\Gamma \vdash_v () : 1]\!] & = & t_{[\![\Gamma]\!]} \\
[\![\Gamma, x : T \vdash_v x : T]\!] & = & \pi_r \\
[\![\Gamma \vdash_v \langle U, V\rangle : S \times T]\!] & = & \langle [\![\Gamma \vdash_v U : S]\!], [\![\Gamma \vdash V : T]\!]\rangle \\
[\![\Gamma \vdash_v \mathrm{in_i(V)} : \mathrm{T_1 + T_2}]\!] & = & [\![\Gamma \vdash_v V : T_i]\!]; \iota_i \\
[\![\Gamma \vdash_v \lambda x.M : S \to T]\!] & = & \Lambda([\![\Gamma, x : T \vdash_c M : T]\!]) \\
 & & \\
[\![\Gamma \vdash_c \mathtt{void}(V) : T]\!] & = & [\![\Gamma \vdash_v V : 0]\!]; i_{\Sigma[\![T]\!]} \\
[\![\Gamma \vdash_c [V] : T]\!] & = & [\![\Gamma \vdash_v V : T]\!]; \eta \\
[\![\Gamma \vdash_c U\ V : T]\!] & = & \langle [\![\Gamma \vdash_v U : S \to T]\!], [\![\Gamma \vdash_v V : T]\!]\rangle; \mathsf{app} \\
[\![\Gamma \vdash_c \mathtt{case}\ V\ \mathtt{as}\ \mathrm{in_1(x).M|in_2(x).N}]\!] & = & \langle \mathsf{id}_{[\![\Gamma]\!]}, [\![\Gamma \vdash_v V : S_1 + S_2]\!]\rangle; [[\![\Gamma, x : S_1 \vdash_c M : T]\!], [\![\Gamma, x : S_2 \vdash_c N : T]\!]] \\
[\![\Gamma \vdash_v \mathtt{match}\ V\ \mathtt{as}(x,y).M : T]\!] & = & \langle \mathsf{id}_{[\![\Gamma]\!]}, [\![\Gamma \vdash_v V : S_1 \times S_2]\!]\rangle; [\![\Gamma, x : S_1, y : S_2 \vdash_c M : T]\!] \\
[\![\Gamma \vdash_c \mathtt{let}\ M = x\ \mathtt{in}\ N : T]\!] & = & \langle \mathsf{id}_{[\![\Gamma]\!]}, [\![\Gamma \vdash_c M : S]\!]\rangle; t_{[\![\Gamma]\!],[\![S]\!]}; [\![\Gamma, x : S \vdash_c M : T]\!]^*
\end{array}
$$

Table 3: Interpretation of $\mathscr{L}$-terms

3.1 Game Semantics

We now review the game semantics of $\mathscr{L}$ and its extensions with references [1] and continuations [6] (to which we refer for further details), in a category of arenas and thread-independent strategies.

An *arena* A is a bipartite labelled forest — a triple $\langle M_A, \vdash_A, \lambda_A\rangle$, where M_A is the set of nodes (moves), $\vdash_A \subseteq M_A \times M_A$ (the *enabling* relation) is the set of edges, and $\lambda_A : M_A \to \{Q, A\}$ is a labelling function which partitions moves as *answers* (A) or *questions* (Q), such that answers are enabled by questions. The set of root nodes of the forest is denoted M_A^I — these are called *initial moves*. Partitioning of M_A into *Player* and *Opponent* moves may be inferred from the requirement that initial moves are Opponent moves, and that Player moves are enabled by Opponent moves and vice-versa.

Key constructions on arenas are:

- The disjoint sum of forests (product): $A \times B = (M_A + M_B, \vdash_A + \vdash_B, [\lambda_A, \lambda_B])$.
- The graft of A onto the roots of B (function space): $A \Rightarrow B = (\bigoplus_{m \in M_B^I} M_A + M_B, (\bigoplus_{m \in M_B^I} \vdash_A) + \vdash_B \cup \{(m, \mathrm{in_m(n)}) \mid \mathrm{m} \in \mathrm{M_B^I}, \mathrm{n} \in \mathrm{M_A^I}\}, [[\lambda_\mathrm{A} \mid \mathrm{m} \in \mathrm{M_B^I}], \lambda_\mathrm{B}])$.

A *legal justified sequence* over the arena A is a finite alternating sequence of moves of A in which each occurrence of a non-initial move n comes with a unique *justification pointer* to a preceding occurrence of a move m which enables n (i.e. such that $m \vdash_A n$).

A strategy σ over an arena A is a non-empty, even-prefix-closed set of even-length alternating justified sequences over A, satisfying:

Determinacy If $sa, sb \in \sigma$ then $b = c$.

Thread-independence If r, s, t are even-length legal sequences such that t is the interleaving of r and s, then $t \in \sigma$ if and only if $r, s \in \sigma$.

The *pending question prefix* (if any) of a justified sequence s is the greatest prefix of s ending with a question which does not occur between an answer and its justifying question in s: i.e.

- $\mathsf{pending}(sq) = q$
- $\mathsf{pending}(sqta) = \mathsf{pending}(s)$, where q justifies a.

A strategy σ is *well-bracketed* if it satisfies the following condition:

Well-Bracketing Any answer-move played by σ is justified by the question pending when it was played — i.e $sqta \in \sigma$ (where a points to q) implies $\mathsf{pending}(sqt) = sq$.

Composition of strategies $\sigma : A \Rightarrow B, \tau : B \Rightarrow C$ is by parallel composition plus hiding of moves in B. $\sigma;\tau = \{s \in L_{A\Rightarrow B} \mid \exists t \in L_{((A\Rightarrow B)\Rightarrow C)}.t\restriction A,B \in \sigma \wedge t\restriction B,C \in \tau \wedge t\restriction A,C = s\}$. This yields a Cartesian closed category $\mathscr{G}$ in which objects are arenas, morphisms from A to B are strategies on $A \Rightarrow B$, and identities are copycat strategies. It has a cartesian closed, wide subcategory $\mathscr{G}^B$ in which morphisms are well-bracketed strategies [1, 6].

3.2 Semantics of $\mathscr{L}$

We interpret $\mathscr{L}$ by exhibiting a strong monad on the category of "pre-arenas" obtained by applying the $\mathsf{Fam}(\_)$ construction (small co-product completion) to $\mathscr{G}^B$ (following [2]). For any category $\mathscr{C}$, $\mathsf{Fam}(\mathscr{G})$ is the category of *set-indexed families* of objects of $\mathscr{C}$, which has as morphisms from $\{A_i \mid i \in I\}$ to $\{B_j \mid j \in J\}$, a pair $\langle f : I \to J, \{\psi_i : A_i \to B_{f(i)} \mid i \in I\}\rangle$ of a re-indexing function and a family of morphisms in $\mathscr{C}$.

If $\mathscr{C}$ is Cartesian closed, then so is $\mathsf{Fam}(\mathscr{C})$:

- $\mathsf{Fam}(\mathscr{C})$ has co-products, given by the disjoint union of indexed families.
- $\mathsf{Fam}(\mathscr{C})$ has products, — $\{A_i \mid i \in I\} \times \{B_j \mid j \in j\}$ is $\{A_i \times B_j \mid \langle i, j\rangle \in I \times J\}$.
- $\mathsf{Fam}(\mathscr{C})$ has exponentials — in particular, for any arena B, exponentials of the singleton family $\{B\}$: $\{A_i \mid i \in I\} \Rightarrow \{B\} = \{\Pi_{i\in I}(A_i \Rightarrow B)\}$.

A justified sequence on $A \Rightarrow B$ is *linear* if every initial move in B justifies exactly one initial move in A. A (thread-independent) strategy $\sigma : A \to B$ is linear if it contains some non-empty sequence, and every sequence $s \in \sigma$ is linear. A *tree arena* is an arena with a unique root (initial move). Let $\mathscr{G}_S$ be the subcategory of $\mathscr{G}^B$ consisting of tree arenas and linear strategies.

Proposition 3.1 *The inclusion of $\mathscr{G}_S$ in $\mathsf{Fam}(\mathscr{G}^B)$ has a left adjoint $\Sigma\_$.*

PROOF: The *lifted sum* [2] of a family of arenas $A = \{A_i \mid i \in I\}$ is the tree ΣA with a single (question) root node, beneath which are answer nodes for each $i \in I$, beneath each of which is the arena A_i:

- $M_{\Sigma A} = (\bigoplus_{i\in I}(M_{A_i} + \{a_i\})) + \{q\}$
- $\lambda^{QA}_{\Sigma A}(q) = Q, \lambda^{QA}_{\Sigma A}(a_i) = A\ i \in I,\ \ \lambda^{QA}_{\Sigma A}(\langle m, i\rangle) = \lambda_{A_i}(m)$
- $* \vdash_{\Sigma A} q \vdash_{\Sigma A} a_i \vdash_{\Sigma A} \langle m, i\rangle$ and $\langle l, i\rangle \vdash \langle n, i\rangle$ where $m \in M^I_{A_i}$ and $l \vdash_{A_i} n$.

There is an evident correspondence between non-empty even-length linear sequences on $A \Rightarrow \Sigma B$ and even-length sequences on $A \Rightarrow B$ yielding an adjunction

$$\frac{\mathscr{G}_S(\Sigma A, B)}{\mathsf{Fam}(\mathscr{G}^B)(A, \{B\})}$$

□

Hence we have a (strong) monad on $\mathsf{Fam}(\mathscr{G}^B)$ sending A to the singleton family $\{\Sigma A\}$ [2], giving a semantics of $\mathscr{L}$. To extend this to a semantics of $\mathscr{L}_{\mathscr{R}}$ it suffices to give the denotation of the non-functional part — the constant $\mathtt{new}_T : \mathtt{var}[T]$ — as a strategy $\mathsf{cell}_A : \Sigma(\Sigma A \times (A \Rightarrow \Sigma 1))$ defined in [1], which takes an argument of type T and behaves as a reference cell initialized with that argument. This yields a computationally adequate semantics of $\mathscr{L}_{\mathscr{R}}$, as proved in [1]:

Proposition 3.2 *$M \Downarrow$ if and only if $[\![M]\!] \neq \bot$.*

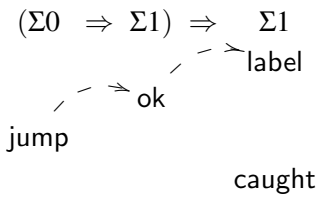
3.3 Semantics of Continuations

We may interpret call-with-current-continuation directly — by identifying its denotation as a strategy, as for new variable declaration — but also indirectly, by *CPS interpretation*. We recall the observation in [6] that relaxing the bracketing condition on strategies renders the lifted sum of games equivalent to a CPS construction. More precisely, let $U_C : \mathsf{Fam}(\mathscr{G}) \to \mathsf{Fam}(\mathscr{G}^B)$ be the identity-on-morphisms functor which acts on arenas by relabelling every answer as a question. Then there is an evident isomorphism of arenas: $U_C(\Sigma A) \cong (\Sigma 0)^{(\Sigma 0)^A}$ yielding an equivalence of the lifting monad (on $\mathsf{Fam}(\mathscr{G}^C)$) to a CPS monad on $\mathsf{Fam}(\mathscr{G})$:

Lemma 3.3 $\Sigma \cdot U_C \cong U_C \cdot (\Sigma 0)^{(\Sigma 0)^-}$

Hence we have a strong monad Σ_- on $\mathsf{Fam}(\mathscr{G})$, equivalent to the one-move-game continuations monad — and thus a semantics of $\mathscr{L}$ in $\mathscr{G}$, such that the inclusion functor $J : \mathscr{G}^B \to \mathscr{G}$ preserves meanings of $\mathscr{L}_{\mathscr{R}}$ terms.

For any game A, the isomorphism from $(A \Rightarrow \Sigma 0) \Rightarrow \Sigma 0$ into ΣA yields a map $\mathsf{callcc}_{A,B} : ((A \Rightarrow \Sigma B) \Rightarrow \Sigma A) \to \Sigma A$ with which to interpret `callcc`. Concretely, this responds to the initial question (label) with a Player question (ok), and to its answer or the subsequent question (throw) with an answer to the initial question (caught) (note that this violates the bracketing condition), and thereafter plays copycat between moves hereditarily enabled by (ok), and those hereditarily enabled by caught. A typical play of $\mathsf{callcc}_{1,0}$ is as follows:



We may prove that our model is sound by using its characterization as a CPS interpretation — either by using an equational theory for CPS models to verify soundness of the reduction rules directly (see [6] for details), or by formalizing it as a CPS translation of $\mathscr{L}_{\mathscr{R}\mathscr{C}}$ into $\mathscr{L}_{\mathscr{R}}$ and showing that it is sound. We give a proof of the latter form in the Appendix, showing that:

Proposition 3.4 *Interpretation of programs as strategies in $\mathscr{G}$ is sound and computationally adequate: $M \Downarrow$ if and only if $[\![M]\!] \neq \bot$.*

4 Control Strategies

We now extend the game semantics of $\mathscr{L}_{\mathscr{R}\mathscr{C}}$ with continuations and exceptions, to interpret $\mathscr{L}_{\mathscr{R}\mathscr{C}\mathscr{E}}$. We retain the interpretation of $\mathscr{L}$-types as (families of) arenas, but change the notion of strategy by adding control-flow information to justified sequences in the form of "control pointers". (These were introduced in [7] in the context of interpreting a call-by-name language without first-class continuations: here, we relax the *weak bracketing condition* imposed in that model.

Definition 4.1 *A* control sequence *s over an arena A is an alternating justified sequence $|s|$ over A, together with a* control pointer *from each question move in $|s|$ either to a unique preceding question (or else to a distinguished root token $*$) — such that Opponent moves point to Player moves or $*$ and Player moves point to Opponent moves.*

We write C_A for the set of control sequences over the arena A. A *control strategy* on A is a non-empty, even-prefix-closed set of even-length control sequences in C_A, satisfying the determinacy and thread-independence conditions.

In order to use our definition of composition for control strategies, we need to define the restriction operator on control sequences to replace "dangling" control pointers, by following back pointers to hidden moves until an unhidden move is reached. Accordingly, we define the set of *open questions* of a control sequence as follows:
$\mathsf{open}(\varepsilon) = \{\}$,
$\mathsf{open}(sqta) = \mathsf{open}(s)$, if a is an answer
$\mathsf{open}(sqtq') = \mathsf{open}(sq) \cup \{sqtq'\}$ if q' is a question with a control pointer to q,
$\mathsf{open}(sq) = \{q\}$, if q points to $*$.

We extend the restriction operation to control sequences by requiring that every move in $s{\restriction}B$ points to the most recent preceding open move which is in B (if any). With this definition of restriction, the original proofs of well-definedness and associativity of parallel composition plus hiding [10] extend straightforwardly to control strategies.

To form a category, we also need to define identity morphisms (and other copycats) as control strategies. Say that a control sequence s satisfies (player) *control locality* if every Player question in s points to the pending question: let $\mathsf{Loc_A}$ be the set of control sequences over A which satisfy this condition. Given a strategy on A, we may define a local control strategy $\widehat{\sigma}$ on A by taking all player-local sequences which correspond to sequences in σ when pointers are ignored i.e. $\widehat{\sigma} = \{s \in Loc_A \mid |s| \in \sigma\}$. (In other words, by decorating sequences in σ by adding control pointers from each Opponent question to some Player question, and from each Player question to its pending question.) We define the identity control strategy to be $\widehat{\mathsf{id}_A}$ (and similarly for the other copycat strategies giving cartesian closed structure). So we may define a cartesian closed category $\mathscr{CG}$ in which objects are arenas, and morphisms from A to B are control strategies on $A \to B$.

We also observe that:

- The operation $\widehat{\_}$ is not functorial: the arenas $\Sigma 1$ and $\Sigma 0 \Rightarrow \Sigma 0$ are isomorphic in $\mathscr{G}$ but not in $\mathscr{CG}$: the composition of the images of these isomorphisms under $\widehat{\_}$ is not $\widehat{\mathsf{id}_{\Sigma 0 \Rightarrow \Sigma 0}}$.
- $\widehat{\_}$ *is* functorial on well-bracketed strategies: there is a faithful, identity-on-objects functor $J : \mathscr{G}^B \to \mathscr{CG}$ sending $\sigma : A \to B$ to $\widehat{\sigma}$.

4.1 Semantics of Exceptions

$\mathscr{CG}$ has structure with which to model $\mathscr{L}$ — (it is a CCC with a strong lifted sum monad $\Sigma_\_$ on $\mathsf{Fam}(\mathscr{CG})$. The functor $J : \mathscr{G}^B \to \mathscr{CG}$ preserves all of this structure, and hence the meaning of $\mathscr{L}$-terms. We interpret new reference declaration, and call-with-current-continuation by decorating the corresponding underlying strategies with control pointers: i.e.

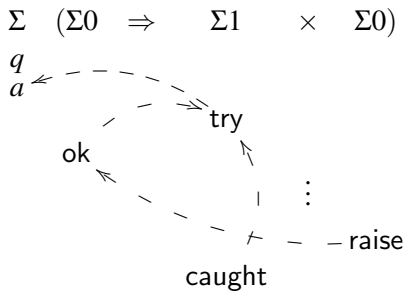
- $[\![\mathtt{new}_T]\!]_C = \widehat{\mathsf{cell}_{[\![T]\!]}}$
- $[\![\mathtt{callcc}]\!]_C = \widehat{\mathsf{callcc}}$

Since cell is well-bracketed, it lies within the domain of the functor J, which therefore preserves the meaning of $\mathscr{L}_{\mathscr{R}}$-terms by definition. As we have noted, $\widehat{\_}$ is not functorial on non-well-bracketed strategies, and hence does not preserve the meaning of $\mathscr{L}_{\mathscr{RC}}$-terms in general.

Exceptions We interpret new-exception declaration as a new-exception strategy $\mathsf{exn}_C : \Sigma((\Sigma 0 \Rightarrow \Sigma 1) \times (\Sigma 0))$. This was defined in a weakly-bracketed setting in [7]: it relies on control pointers to determine the current exception handler. Its behaviour can be described as follows:

- Answer the initial (Opponent) question.
- If Opponent plays the initial move on the left (try) then respond with a question (ok).
- If Opponent plays the initial move on the right (throw), then answer the most recent open initial question on the left (caught). If there are no such moves, do nothing, representing divergence caused by an uncaught exception.

In other words, exn_C consists of the legal control sequences of the form $qa(((\mathsf{tryok})^*(\mathsf{throwcaught})^*)^*$ such that each caught move is justified by the most recent open try move. Here is a typical play:



Proving soundness for this model directly is difficult due to the implicit nature of its representations of continuations and exceptions. Instead, we will provide an alternative characterization of exn by relating it to an exceptions monad.

5 A Monadic Effect Semantics

In this section, we construct a semantics of $\mathscr{L}_{\mathscr{RCE}}$ corresponding to exceptions and continuation-passing-style interpretation. This has the advantage of relating these control effects to well-understood structure, at the cost of a less direct and concrete interpretation of terms. Observe that since the (strong) exceptions monad $\_ + E$ distributes over any other monad, we have a monad $\Sigma(\_ + 1)$ on $\mathsf{Fam}(\mathscr{G})$. By Lemma 3.3, this is equivalent to the exceptions-and-continuations monad $R^{R^{-}+E}$, for the answer object $R = \Sigma 0$.

In fact, we will describe the semantics of $\mathscr{L}_{\mathscr{RCE}}$ in a category of "exception arenas" in which raising and propagating the global exception is represented by playing explicit "exception moves". The lifting monad in this model is equivalent to the above exceptions-and-continuations monad, giving a route to establishing its soundness, whilst it may be related to the control games interpretation by replacing runs of exception moves with control pointers.

Definition 5.1 *An exception arena is a pair (A, e_A) of an arena together with a function $e_A : \{m \in M_A \mid \lambda(m) = Q\} \to \{m \in M_A \mid \lambda(m) = A\}$ associating each question q with a unique "exception answer", which is a child of q — i.e $q \vdash e_A(q)$ — and a leaf of A.*

Exception moves correspond to a single, global exception: playing an exception answer in response to a non-exception move corresponds to *raising* this exception, playing the pending exception answer in response to an exception move corresponds to *propagating* it, and playing a non-exception move in response to an exception move corresponds to *handling* it.

Observe that if (A, e_A) and (B, e_B) are exception arenas, then $(A \Rightarrow B, \bigoplus_{m \in M_B^I} e_A + e_B$ and $(A \times B, e_A + e_B)$ are exception arenas. Hence we may define:

- a Cartesian closed category $\mathscr{G}^E$, which has exception-arenas as objects and unbracketed strategies on $A \Rightarrow B$ as morphisms from A to B.
- A fully faithful (identity on morphisms) cartesian closed functor $U_E : \mathscr{G}^E \to \mathscr{G}$ which forgets the exception answer labelling.

Given a family of exception-arenas $A = \{(A_i, e_i) \mid i \in I\}$, define $\Sigma_E A = (\Sigma(\{A_i \mid i \in I\} + 1), \bigoplus_{i \in I} e_i \cup \{q, \mathtt{in_r}(a)\})$ — i.e. the exception arena given by extending $\Sigma\{A_i \mid i \in I\}$ with an additional exception answer move $\mathtt{in_r}(a)$ to the initial question. By definition, we have:

Lemma 5.2 $U_E(\Sigma_E B) = \Sigma(U_E(B) + 1)$.

Hence by full faithfulness of U_E, we therefore have a strong monad Σ_E on $\mathsf{Fam}(\mathscr{G}^E)$ such that $U_E \cdot \Sigma_E = \Sigma(\_ + 1) \cdot U_E$, giving a semantics for the type-theory $\mathscr{L}$ in $\mathscr{G}^E$. Note that by Lemma 3.3, Σ_E is equivalent to the exceptions-with-continuations monad $R^{R^{-}+1}$, where R is the one-move game — i.e. $U_C \cdot U_E \cdot \Sigma_E \cong R^{R^{-}+1} \cdot U_C \cdot U_E$.

We may interpret continuations and references in this model by adding exception-answers to our existing denotations for these features. Given an exception-arena A, let $K(A)$ be the arena obtained by erasing all of the exception-answers in A — i.e. $M_{K(A)} = M_A - \{e(q) \mid \lambda(q) = Q\}$. Say that an even-length legal sequence s on A is exception-local if:

- Player always propagates exception moves: if $te(q)m \sqsubseteq s$ is even-length, then $m = e(q')$, where q' is the pending question of $te(q)$.
- Player never raises an exception: if $tmn \sqsubseteq s$ is even-length, and m is not an exception-move, then n is not an exception move.

Given an exception-local sequence, let $\overline{s}$ be the legal sequence on $K(A)$ obtained by erasing exception moves in s. Let the *exception-completion* of a strategy σ on $K(A)$, be the set $\widetilde{\sigma}$ of exception-local sequences on A such that $\overline{s} \in \sigma$. (Note that this is deterministic and thread-independent.) Hence we may define the denotation of `callcc` and `new` as the exception-completions of their denotations in $\mathscr{G}$.

To interpret exception declaration and handing, first note that we have evident *raise* and *handle* strategies on $1 \to \Sigma_E 1$ and $\Sigma_E 0 \to \Sigma_E 1$ which raise and handle the *global* exception by playing the exception-answer to the initial question and respond to Opponent's playing of an exception answer by playing a "non-exception-answer", respectively. The interpretation of $\mathtt{new\_exn} : ((1 \to 0) \to 1) \times (1 \to 0)$ as a strategy $\mathsf{exn}_E : ((\Sigma_E 0 \Rightarrow \Sigma_E 1) \times \Sigma_E 0$ combines these behaviours with local state: the handle method handles the global exception if the raise method has been used to raise a global exception which has not yet been handled, and propagates it otherwise. Formally, define exn_E to consist of all plays of the form $qa(\mathsf{handlok})^*(\mathsf{raise} \cdot e[\mathsf{raise}]))^*(e[\mathsf{ok}]\mathsf{caught})^*$ which are Player-well-bracketed. Example plays are given in Figure 1.

5.1 Soundness

We may prove soundness and adequacy of our exceptions monad model by showing that it corresponds to an *exception-passing-style* translation $(\_)^E$ from $\mathscr{L}_{\mathscr{RCE}}$ to $\mathscr{L}_{\mathscr{RC}}$. This acts on types as follows:

- $0^E = 0$, $1^E = 1$,
- $(S \times T)^E = S^E \times T^E$,

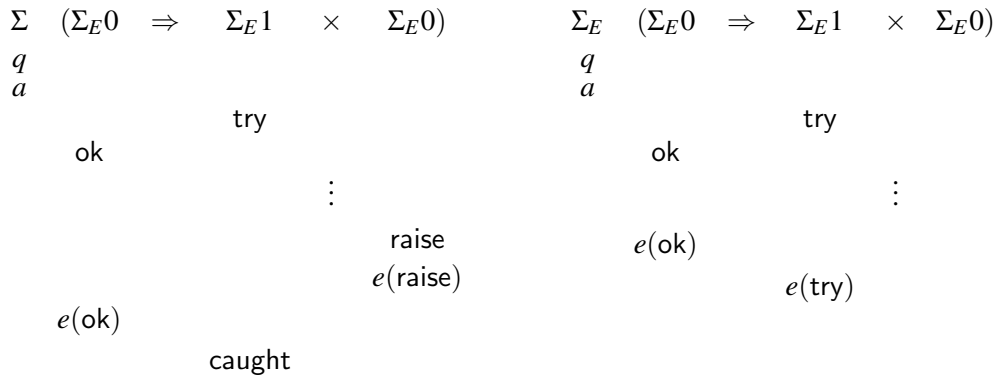


Figure 1: Plays of exn_E

- $(S+T)^E = S^E + T^E$,
- $(S \to T)^E = S^E \to (T^E + 1)$.

Translation of computations $x_1 : S_1, \ldots, x_n : S_n \vdash M : T$ as computations $x_1 : S_1^E, \ldots, x_n : S_n^E \vdash_c M^E : T^E + 1$ and values $x_1 : S_1, \ldots, x_n : S_n \vdash V : T$ as values $x_1 : S_1^E, \ldots, x_n : S_n^E \vdash_E V^E : T^E$ is given in Table 4. The translation of $\mathscr{L}_{\mathscr{R}\mathscr{C}}$ arises straightforwardly from the strong monad structure, so we focus on the interpretation of *local* exceptions — specifically, the new-exception declaration. Right injection and pattern matching may be used to define evident operations which raise and handle the single *global* exception, respectively. But how can we use these to construct an object whose methods raise and handle a *local* exception? The answer is: we use the local state in our underlying model/metalanguage. Our exception has as its internal state a Boolean variable e acting as a flag. The raise method for the local exception object sets e and raises the global exception. The handle method for the object handles the global exception, tests e and resets it if it is set (i.e. e had been raised and has now been handled) or re-raises the global exception if it is not set (i.e. some other exception was raised and now needs to be propagated). So we have two methods:

- $\mathtt{raise}(e) = \lambda z.e := \mathtt{tt}; \mathrm{in}_2(()) : (1 \to 0)^{\mathrm{E}}$
- $\mathtt{handle}(e) = \lambda f.(f\,()); \mathtt{If}\ \mathtt{deref}(e)\ \mathtt{then}\ (e := \mathbf{ff}; \mathrm{in}_2(()))\ \mathtt{else}\ \mathrm{in}_2(()) : ((1 \to 0) \to 1)^{\mathrm{E}}$

New-exception declaration simply aggregates these methods and hides the internal state e. We establish that the interpretation of local exceptions via translation into $\mathscr{L}_{\mathscr{R}\mathscr{C}}$ is sound.

Lemma 5.3 *$M \Downarrow$ if and only if $M^E \Downarrow$.*

PROOF: The key step is to break the exception-propagation rule down to propagate exceptions past each `let`-context and non-matching handler. □

We have already noted that the forgetful functor U_E sends Σ_E to the (lifted) exceptions monad $\Sigma(\_ + 1)$. This correspondence extends to the interpretation of $\mathscr{L}_{\mathscr{R}\mathscr{C}}$-types and terms.

Lemma 5.4 *For any $\mathscr{L}$-term, $\Gamma \vdash M : T$, $U_E([\![\Gamma \vdash M : T]\!]) = [\![M^E]\!]$.*

PROOF: For types, and terms of $\mathscr{L}_{\mathscr{R}\mathscr{C}}$, this follows from equivalence of Σ_E and $\Sigma_- + 1$ via U_E. Thus, it remains to observe that these interpretations agree on the interpretation of new exception declaration — i.e. $[\![\lambda().\mathtt{new}\, x := \mathbf{ff}.[\mathrm{in}_1(\langle \mathbf{handle}(\mathrm{e}), \mathtt{raise}(\mathrm{e})\rangle)]]\!] = \mathrm{U_E}(\mathsf{exn}_{\mathrm{E}})$. □

Hence we have soundness and adequacy for the exception-arena semantics.

Proposition 5.5 *$M \Downarrow$ if and only if $[\![M]\!]_E \neq \bot$*

- $(x)^E = x$
- $[V]^E = [\mathrm{in}_1(\mathrm{V^E})]$
- $(\mathtt{let}\, M = x\, \mathtt{in}\, N)^E = \mathtt{let}\, M^E = y\, \mathtt{in}\, \mathtt{case}\, y\, \mathtt{as}\, \mathrm{in}_1(\mathrm{x}).\mathrm{N^E}|\mathrm{in}_2(\mathrm{x}).\mathrm{in}_2(())$
- $(\lambda x.M)^E = \lambda x.M^E,\ \ (U\ V)^E = U^E\ V^E$
- $(\mathtt{match}\, V\, \mathtt{as}\, (x,y).M)^E = \mathtt{match}\, (x,y)\, \mathtt{as}\, V^E\, \mathtt{in}\, M^E,\quad \langle U,V\rangle^E = \langle U^E, V^E\rangle,$
- $()^E = (),\quad \mathtt{void}(V)^E = \mathtt{void}(V^E)$
- $\mathrm{in_i}(\mathrm{V})^\mathrm{E} = \mathrm{in_i}(\mathrm{V^E}),\ \ (\mathtt{case}\, V\ \mathtt{as}\, \mathrm{in}_1(\mathrm{x}).\mathrm{M}|\mathrm{in}_2(\mathrm{x}).\mathrm{N})^\mathrm{E} = \mathtt{case}\, \mathrm{V^E}\, \mathtt{as}\, \mathrm{in}_1(\mathrm{x}).\mathrm{M^E}|\mathrm{in}_2(\mathrm{x}).\mathrm{N^E}$
- $\mathtt{callcc}(V)^E = \mathtt{callcc}(\lambda k.V^E\ \lambda x.k\, \mathrm{in}_1(\mathrm{x}))$
- $\mathtt{new}^E = \lambda().\mathtt{new}\, a\, \mathtt{in}\, [\mathrm{in}_1(\mathrm{a})]$
- $\mathtt{new\_exn}^E = \lambda().\mathtt{new}\, x := \mathbf{ff}.[\mathrm{in}_1(\langle \mathtt{handle}(\mathrm{e}), \mathtt{raise}(\mathrm{e})\rangle)]$

Table 4: Exception-passing translation

6 Relating Control Games to the Exception Monad

We now relate the monadic and control-games interpretations of $\mathscr{L}_{\mathscr{R}\mathscr{C}\mathscr{E}}$ by establishing a correspondence between the two models: a meaning-preserving functor into $\mathscr{C}\mathscr{G}$ from a subcategory of $\mathscr{G}^E$ consisting of exception-arenas and *exception-propagating strategies*.

A sequence s over A is *exception-propagating* if whenever Opponent raises an exception, Player always propagates it by playing the exception-answer to the pending question, and vice versa. Formally, define the set of exception-propagating sequences to be the least set of justified sequences such that:

- The empty sequence is exception-propagating,
- If s is exception-propagating, and m is not an exception move, then sm is exception-propagating.
- If s (of even length) is exception-propagating then $se(q)e(q')$ is exception-propagating, where q' is the pending question of $se(q)$.

We write EP_A for the set of exception-propagating control sequences on A. Recall that we defined $K(A)$ to be the arena obtained by erasing all of the exception-answers in A. Given $s \in EP_A$ we define a control sequence $K(s)$ on $K(A)$ by:

- First, adding a control pointer from each question to its pending question (or the token $*$ otherwise).
- Then, deleting all exception answers.

This is a well-defined control sequence; it is alternating since if s is exception-propagating then all exception-moves in s come in adjacent pairs. Control pointers alternate in polarity since the pending question is always of opposite polarity to the move about to be played (and there is always a pending question at any Player move). For an example, the first typical play given for the new-exception strategy exn_E (Fig. 1) is transformed to the typical play given for the corresponding control strategy exn_C

Extend the definition of K to all justified sequences on the exception-arena A by letting $K(s) = K(t)$, where t is the greatest exception-propagating prefix of s. A *strategy* σ on A is *exception-propagating* if $K(\sigma) = \{K(s) \mid s \in \sigma\}$ is a well-defined (thread-independent) control strategy. In other words:

- $K(\sigma)$ consists of even-length sequences — σ always propagates exceptions raised by Opponent.

- $K(\sigma)$ is even-branching — σ ignores exceptions raised by Opponent once they have been handled (but not their effect on the exception handling context).

We show that this is a compositional property of strategies, and that the action of K is functorial, based on the following lemma:

Lemma 6.1 *Given $s \in L_{(A\to B)\to C}$, if $s\restriction A,B$ and $s\restriction B,C$ are legal and exception-propagating, then:*

- *$s\restriction A,C$ is exception-propagating.*
- *$K(s)\restriction A,C = K(s\restriction A,C)$.*

PROOF: Since the $s\restriction A,B$ and $s\restriction B,C$ are exception-propagating, any runs of exception-answers occur in even-length blocks in s, and erasing the part in B leaves an exception-propagating sequence.

We show by induction on the length of s that the pending question in $s\restriction A,C$ is the pending question in the relevant fragment $s\restriction A,B$ or $s\restriction B,C$, and so control pointers in $K(s)\restriction A,C$ and $K(s\restriction A,C)$ agree. □

Based on this lemma, we show:

Proposition 6.2 *The composition of exception-propagating strategies is exception-propagating.*

It is straightforward to verify that the identity strategy is exception propagating, with $K(\mathsf{id}) = \mathsf{id}$. Hence:

- Exception-propagating strategies form a lluf subcategory $\mathscr{G}^{EP}$ of $\mathscr{G}^{E}$.
- K acts as a functor from $\mathscr{G}^{EP}$ to $\mathscr{CG}$.

Evidently, K preserves Cartesian closed structure, and $\Sigma_E \cdot K \cong \Sigma \cdot K$. So for $\mathscr{L}$-types we have $K([\![T]\!]_E) \cong [\![T]\!]_C$. Moreover, if we apply K to the exception-completion of a strategy, this is equivalent to decorating with control-pointers to the pending moves — i.e. $K(\widetilde{\sigma}) = \widehat{\sigma}$. So K preserves the meaning of $\mathscr{L}_{\mathscr{RC}}$-terms. It remains to check that this is the case for exception declaration.

Lemma 6.3 *exn_E is exception-propagating, and $K(\mathsf{exn}_E) \cong \mathsf{exn}_C$.*

PROOF: Recall that exn_E is the set of well-bracketed plays $qa(\mathsf{handleok})^*(\mathsf{raise}\cdot e[\mathsf{raise}]))^*(e[\mathsf{ok}]\mathsf{caught})^*$. This is evidently exception-propagating, and erasing the exception moves on exception-propagating plays leaves a control sequence of the form $qa(((\mathsf{tryok})^*(\mathsf{throwcaught})^*)^*$, where the pending try move in the original sequence becomes the target of a control pointer and hence the most recent open handler as required. □

Thus we have shown that:

Proposition 6.4 *Every $\mathscr{L}_{\mathscr{RCE}}$-term M denotes an exception-propagating strategy such that $K([\![M]\!]_E) = [\![M]\!]_C$.*

and hence established soundness and adequacy for the exception-arena semantics.

Proposition 6.5 *$M \Downarrow$ if and only if $[\![M]\!]_C \neq \bot$.*

7 Full Abstraction

Having proved soundness for the control strategy model using algebraic methods (correspondence with the monad model), we may establish full abstraction via reduction (by factorization) to the definability result for the original model of $\mathscr{L}_{\mathscr{R}}$.

Recall that a (thread-independent) strategy σ is compact (in the inclusion order) if the set of *well-opened* sequences (those having a unique initial move) in σ is finite. The following result is proved (by factorisation) in [1].

Proposition 7.1 *For any type $\mathscr{L}$-type T, every compact well-bracketed strategy on $\Sigma[\![T]\!]$ is the denotation of a $\mathscr{L}_{\mathscr{R}}$ term $M_\sigma : T$.*

Since $J : \mathscr{G}^B \to \mathscr{CG}$ preserves the meaning of $\mathscr{L}_{\mathscr{R}}$-terms, every compact strategy in the image of J is definable — i.e. all compact, local well-bracketed strategies which also satisfy the following condition:

Control blindness $|\sigma| = \{|s| \mid s \in \sigma\}$ is a deterministic strategy on A.

Corollary 7.2 *Every compact local, well-bracketed and control-blind strategy over an $\mathscr{L}$ type-object is definable as a term of $\mathscr{L}_{\mathscr{R}}$.*

We now factorize any compact control-blind strategy into the composition of a definable strategy with the denotation of `callcc`.

Lemma 7.3 *For any (compact) control-blind strategy $\sigma : 1 \to A$ there is a (finitary) local, well-bracketed strategy $\widetilde{\sigma} : ((\Sigma 0 \Rightarrow \Sigma 1) \Rightarrow \Sigma 1) \to A$ such that $\Lambda(\mathsf{callcc}_{1,0}); \widetilde{\sigma} = \sigma$.*

PROOF: This is essentially the factorization given in [5]: we define a map $\widetilde{\_}$ from control sequences to Player well-bracketed sequences in $L_{((\Sigma 0 \Rightarrow \Sigma 0) \Rightarrow \Sigma 1) \Rightarrow A}$ which interjects $\mathsf{label}, \mathsf{ok}$ after each Opponent question, and blocks of $\mathsf{jump}, \mathsf{caught}$ moves before each Player answer, so that any intervening questions are closed.

□

So it remains to show that control-blind strategies may be factorized as the composition of a local control strategy with the strategy exn.

Lemma 7.4 *For any (compact) control strategy $\sigma : 1 \to A$ there is a (compact) control-blind strategy $\widehat{\sigma} : [\![\mathtt{exn}]\!] \to A$ such that $\mathsf{exn}; \widehat{\sigma} = \sigma$*

PROOF: We define a map $\widehat{\_}$ from control sequences on A to control sequences on $\mathtt{exn} \to A$ which makes all control pointers on O-moves manifest. It raises an exception before each Opponent move, and handles it after each O-move. The handler which catches the raised exception is determined by the control pointers from O-moves in s, so that $|\widetilde{s}| = |\widetilde{t}|$ implies $s = t$. Since $\widetilde{s}{\restriction}A = s$, and $\widetilde{s}{\restriction}[\![\mathtt{exn}]\!] \in \mathsf{exn}$, we have $\mathsf{exn}; \widehat{\sigma} = \sigma$ as required □

For any compact strategy, $\sigma : I \to \Sigma[\![T]\!]$, the control-blind strategy $\widetilde{\sigma}$ is definable as a term $x : \mathtt{exn} \vdash M$ and thus σ is definable as $\mathtt{let}\, x = \mathtt{new\_exn}\, \mathtt{in}\, M$. The proof of full abstraction based on finite definability is now standard.

Theorem 7.5 *The model of $\mathscr{L}_{\mathscr{RCE}}$ in $\mathsf{Fam}(\mathscr{CG})_\Sigma$ is (inequationally) fully abstract — i.e. for any $\mathscr{L}$-type T and any terms $M, N : T$, $[\![M]\!]_C \subseteq [\![N]\!]_C$ if and only if $M \lesssim N$.*

8 Conclusions and Further Directions

Model checking exceptions Giving different representations of exceptions in games models may be useful in the developing field of program-verification based on semantic games. For example, we may observe that the set of exception-propagating sequences over a finite alphabet (with a specified subset of distinguished exception tokens) is regular, giving a way of extending results characterizing finite-state representable fragments of imperative languages to include local exceptions. On the other hand control pointers describe control flow (and, in particular, exception handling points) directly, and so adding them to game semantic approaches to control flow analysis [9] offers the possibility of reasoning about e.g. exception safety.

Delimited Control Further instances of delimited continuations such as locally declared, dynamically bound *prompts* [3] could be modelled by a similar analysis relating CPS interpretation to the stateful behaviour in games models.

Good Variables Languages such as ML and Java have explicit exception types, so that an object of exception type must behave as an exception, whereas there is clearly no such constraint on objects of the product type which we have used as an exception type. Extending our full abstraction results to such languages is liable to require some characterization of such behavioural constraints. This problem is analogous to the "good variable" problem for references, and we may look to research in this area for approaches to model "good exceptions" [12]. Implementing *wildcard handling* (e.g. Java's `finally`) becomes straightforward when exceptions are passed as names through an exceptions monad, although a wildcard handler typically cannot trap an exception and then discover its name, and so a model should reflect this constraint.

References

[1] S. Abramsky, K. Honda & G. McCusker (1998): *A fully abstract games semantics for general references*. In: *Proceedings of the 13th Annual Symposium on Logic In Computer Science, LICS '98*, IEEE Press, doi:10.1109/LICS.1998.705669.

[2] S. Abramsky & G. McCusker (1998): *Call-by-value Games*. In M. Neilsen & W. Thomas, editors: *Proceedings of CSL '97*, Springer-Verlag, pp. 1–17, doi:10.1007/BFb0028004.

[3] C. Gunter, D. Rémy, and J. Riecke (1995): *A generalization of exceptions and control in ML like languages*. In: *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pp. 12–23, doi:10.1145/224164.224173.

[4] J. M. E. Hyland, P. B. Levy, G. D. Plotkin & J. Power (2007): *Combining Algebraic effects with continuations*. *Theoretical Computer Science* 375(1-3), pp. 20–40, doi:10.1016/j.tcs2006.12.026.

[5] J. Laird (1997): *Full abstraction for functional languages with control*. In: *Proceedings of the Twelfth International Symposium on Logic In Computer Science, LICS '97*, IEEE Computer Society Press, doi:10.1109/LICS.1997.614931.

[6] J. Laird (1998): *A Semantic Analysis of Control*. Ph.D. thesis, Department of Computer Science, University of Edinburgh.

[7] J. Laird (2001): *A fully abstract game semantics of local exceptions*. In: *Proceedings of LICS '01*, IEEE Computer Society Press, doi:10.1109/LICS.2001.932487.

[8] J. Laird (2002): *Exceptions, Continuations and Macro-expressiveness*. In: *Proceedings of ESOP '02, LNCS* 2305, Springer, doi:10.1007/3-540-54927-8_10.

[9] P. Malacaria and C. Hankin (1998): *Generalised Flowcharts and Games*. In: *Proceedings of the 25th International Colloquium on Automata, Langugages and Programming*, doi:10.1007/BFb0055067.

[10] G. McCusker (1996): *Games and full abstraction for a functional metalanguage with recursive types*. Ph.D. thesis, Imperial College London. Published by Cambridge University Press.

[11] E. Moggi (1988): *Computational Lambda-Calculus and monads*. Technical Report ECS-LFCS-88-66, University of Edinburgh Department of Computer Science.

[12] Nikos Tzevelekos (2008): *Full abstraction for nominal exceptions*. Proc. Games and Logic in Programming Languages.

- $(x)^C = x$, $(\mathtt{let}\, x = M\, \mathtt{in}\, N)^C = \lambda\kappa.M^C\, \lambda m.\mathtt{let} x = m\, \mathtt{in}\, (N^C\, \kappa)$
- $[V]^C = \lambda\kappa.\kappa\, V^C$
- $(\lambda x.M)^C = \lambda z.\mathtt{match}\, (x,\kappa)\, \mathtt{as}\, x\, \mathtt{in}\, M^C\, \kappa$, $(U\, V)^C = U^C\, V^C$
- $\langle U,V\rangle^C = \langle U^C, V^C\rangle$ $(\mathtt{match}\, (x,y)\, \mathtt{as}\, V.M)^C = \lambda\kappa.\mathtt{match}\, (x,y)\, \mathtt{as}\, V^C.M^C\, \kappa$
- $()^C = ()$, $\mathtt{void}(V)^C = \lambda\kappa.\kappa\, \mathtt{void}(V^C)$.
- $\mathrm{in}_2(\mathrm{U})^{\mathrm{C}} = \mathrm{in}_2(\mathrm{V}^{\mathrm{C}})$, $\mathrm{in}_1(\mathrm{V})^{\mathrm{C}} = \mathrm{in}_1(\mathrm{V}^{\mathrm{C}})$,
  $(\mathtt{case}\, V\ \mathtt{as}\, \mathrm{in}_1(\mathrm{x}).\mathrm{M}|\mathrm{in}_2(\mathrm{x}).\mathrm{N})^{\mathrm{C}} = \mathtt{case}\, \mathrm{V}^{\mathrm{C}}\, \mathtt{as}\, \mathrm{in}_1(\mathrm{x}).\mathrm{M}^{\mathrm{C}}|\mathrm{in}_2(\mathrm{x}).\mathrm{N}^{\mathrm{C}}$
- $\mathtt{new}^C = \lambda\kappa.\mathtt{new}\, a\, \mathtt{in}\, \kappa\, \langle \lambda x.\mathtt{fst}(x)\, (a := \mathtt{snd}(x)), \lambda y.\mathtt{snd}(y)\, \mathtt{deref}(a)\rangle$
- $\mathtt{callcc}(V)^C = \lambda\kappa.V^C\, \langle k, \lambda x.\mathtt{match}\, (y,z)\, \mathtt{as}\, x\, \mathtt{in}\, k\, x\rangle$

Table 5: CPS translation of exception-free terms

Appendix: Soundness for $\mathscr{L}_{\mathscr{R}\mathscr{C}}$ via CPS translation

We give an interpretation of $\mathscr{L}_{\mathscr{R}\mathscr{C}}$ in $\mathscr{L}_{\mathscr{R}}$ — a CPS translation corresponding to the action of the CPS monad on our denotational model. This acts on types as follows:

- $0^C = 0$, $1^C = 1$,
- $(S+T)^C = S^C + T^C$,
- $(S\times T)^C = S^C \times T^C$,
- $(S\to T)^C = (S^C \times (T^C \to 0)) \to 0$.

Values $x_1 : S_1, \ldots, x_n : S_n \vdash_v V : T$ are translated as values $x_1 : S_1^C, \ldots, x_n : S_n^C \vdash_v V^C : T^C$ and computations $x_1 : S_1, \ldots, x_n : S_n \vdash_c M : T$ are translated as *values* $x_1 : S_1^C, \ldots, x_n : S_n^C \vdash_v M^C : (T^C \to 0) \to 0$ as defined in Table 3.

Extending to $\mathscr{L}_{\mathscr{C}}^{\#}$ by setting $\#(M)^C = \lambda\kappa.M^C\, \tau$ (where $\tau : 1 \to 0$ is a variable representing the top-level continuation), we may show that reduction of a term tracks that of its translation:

Proposition 8.1 *For any program $M : 1$, $M \Downarrow$ if and only if $M^C\, \tau \longrightarrow \tau\, ()$*

We note that CPS interpretation corresponds (up to isomorphism) to the interpretation of $\mathscr{L}_{\mathscr{R}\mathscr{C}}$-types and terms:

Proposition 8.2 *For any $\mathscr{L}$-type T, there is an isomorphism of arenas $\phi_T : U_C([\![T]\!]) \cong [\![T^C]\!]$ such that for any $\mathscr{L}_{\mathscr{R}\mathscr{C}}$-term, $\Gamma \vdash M : T$, $U_C([\![\Gamma \vdash M : T]\!]); \phi_T = [\![M^C]\!]$.*

PROOF: The key type-constructor is the function type: we have $U_C([\![S\to T]\!]) = U_C([\![S]\!]) \Rightarrow U_C(\Sigma[\![T]\!]) \cong [\![S^C]\!] \Rightarrow R^{R^{[\![T^C]\!]}} = ([\![S^C]\!] \times [\![(T^C]\!]) \Rightarrow R) \Rightarrow R \cong [\![(S^C \times (T^C \to 0)) \to 0]\!] \cong (S\to T)^C$. □

Hence interpretation in the games model is sound and adequate:

Proposition 8.3 *$M \Downarrow$ if and only if $[\![M]\!] \neq \bot$.*