Combining and Relating Control Effects and their Semantics

J. 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 implementing powerful constructs such as resumable exceptions or prompts. 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 exception and continuation and continuation passing interpretations.

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 exception and continuation monads, 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 [9].) 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].

To appear in EPTCS.

Exceptions and continuations also provide a test case for semantic theories of combining algebraic effects, studied in detail in [5]. Here we shall simply use the fact that there is a distributive law of the monad $\_ + E$ (exceptions) over $\mathscr{R}^{\mathscr{R}^-}$ (continuations) (which exist for objects E and $\mathscr{R}$ whenever the relevant categorical constructions do), since the exceptions monad distributes over any other monad. Thus we have an exceptions-and-continuations monad $\mathscr{R}^{\mathscr{R}^{-+E}}$.

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 $\mathscr{R}^{\mathscr{R}^-}$, where $\mathscr{R}$ is the "one-move game", giving both direct and indirect (continuation-passing style) interpretations of call/cc [7].

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 $\mathscr{R}^{\mathscr{R}^{-+1}}$[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 [8], 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 fragment of New Jersey SML. The core of the language, $\mathscr{L}$, is a simply-typed call-by-value λ-calculus based on the computational λ-calculus [12].

Types 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:
$U,V := \quad x \mid () \mid \langle U,V \rangle \mid \mathtt{in_l}(U) \mid \mathtt{in_r}(V) \mid \lambda x.M$

[1] Another approach in [13] also uses an exceptions monad on a category of nominal games — here we focus on more concrete models with implicit state.

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

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

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

$$\frac{\Gamma\vdash_v U{:}S}{\Gamma\vdash_v \mathtt{in_l}(U){:}S{+}T} \quad \frac{\Gamma\vdash_v V{:}T}{\Gamma\vdash_v \mathtt{in_r}(V){:}S{+}T} \qquad \frac{\Gamma\vdash_v V{:}S{+}S' \quad \Gamma,x{:}S\vdash_c M{:}T \quad \Gamma,x{:}S'\vdash_c N{:}T}{\Gamma\vdash_c \mathtt{case}\, V\,\mathtt{as}\,\mathtt{in_l}(x).M|\mathtt{in_r}(x).N{:}T}$$

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

Table 1: Typing Judgements for Computations and Values

Computations are given by the grammar:

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

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.

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 exceptions.

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.

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

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 of $\mathscr{L}_{\mathscr{R}}$ with references but without continuations or exceptions (i.e. 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 $(\mathtt{set}(a), !a)$.
- An unbounded set of pairs of constants $(\mathtt{throw}(e), \mathtt{catch}(e))$.
- An operation $\#\_$ taking computations of type 1 to computations of type T.

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

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

$E_h[\_]$ denotes an evaluation context without a $\mathtt{catch}(h)\ \lambda x.\_$ 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 $\mathscr{S}$, 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 $\mathtt{exn}[T]$ 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 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.

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

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

First, we fix what we mean by a model of the type-theory $\mathscr{L}$ (essentially, a model of the computational λ-calculus [12]): a category $\mathscr{C}$ with finite products and coproducts (including terminal and initial objects) and a strong monad Σ on $\mathscr{C}$ such that for any A and B in $\mathscr{C}$, the exponential $A \Rightarrow \Sigma B$ exists.

We may interpret terms and types of $\mathscr{L}$ based on the semantics of the computational λ-calculus: i.e.

- *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 $[\![S \to T]\!] = [\![S]\!] \Rightarrow \Sigma[\![T]\!]$.
- *Values* $\Gamma \vdash_v V : T$ are interpreted as morphisms from $[\![\Gamma]\!]$ to $[\![T]\!]$ in $\mathscr{C}$.
- *Computations* $\Gamma \vdash_c M : T$ are interpreted as morphisms from $[\![\Gamma]\!]$ to $\Sigma[\![T]\!]$ in $\mathscr{C}$.

3.1 Game Semantics

We now review the game semantics of $\mathscr{L}$ and its extension with references, based on [1] (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 partitioning of M_A into Player and Opponent moves may be inferred from the requirement that root nodes (initial moves) are Opponent moves. Root nodes of the forest are called *initial moves*.

Key constructions on arenas are the *product* — the disjoint sum of forests — and the function-space $A \Rightarrow B$, which *grafts* copies of the forest of moves of A below each initial move of B (see e.g. [4]).

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 $m \vdash_A n$). 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 *basic* 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 $r, s \in \sigma$.

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 $\mathsf{pending}(sqt) = sq$.

We will say that a strategy not satisfying the well-bracketing condition is *unbracketed*.

Composition of strategies $\sigma : A_1 \Rightarrow A_2, \tau : A_2 \Rightarrow A_3$ is by parallel composition plus hiding of moves in A_2. $\sigma;\tau = \{s \in L_{A \Rightarrow B} \mid \exists t \in L_{((A \Rightarrow B) \Rightarrow C)}.\, t \upharpoonright A, B \in \sigma \wedge t \upharpoonright B, C \in \tau \wedge t \upharpoonright A, C = s\}$. 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 [1].

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 $\mathscr{G}$ (following [2]). $\mathsf{Fam}(\mathscr{G})$ is the category of *set-indexed families* of arenas, 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 $\mathscr{G}$.

We note the following structure:

- $\mathsf{Fam}(\mathscr{G})$ has co-products, given by the disjoint union of indexed families.
- $\mathsf{Fam}(\mathscr{G})$ has products — $\{A_i \mid i \in I\} \times \{B_j \mid j \in j\}$ is $\{A_i \times B_j \mid \langle i, j\rangle \in I \times J\}$.
- $\mathsf{Fam}(\mathscr{G})$ 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 $\mathscr{G}$ consisting of tree arenas and linear strategies.

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

PROOF: The *lifted sum* 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 (see [2]). 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

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

□

Hence we have a (strong) monad on $\mathsf{Fam}(\mathscr{G})$ 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 $\mathscr{L}_{\mathscr{R}}$ [1]:

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

4 Control Strategies

We now extend the game semantics of $\mathscr{L}_{\mathscr{R}}$ with continuations and exceptions, to interpret $\mathscr{L}_{\mathscr{RCE}}$. We retain the interpretation of $\mathscr{L}$-types as (families of) arenas, but change the notion of strategy by both *relaxing* constraints on control flow (the bracketing condition) and *adding* control-flow information to justified sequences in the form of "control pointers".

Definition 4.1 *A* control sequence *[8] 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 [11] 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 basic strategy on A, we may define a local strategy $\widehat{\sigma}$ on A by taking all player-local sequences which

correspond to sequences in σ when pointers are ignored i.e. $\widehat{\sigma} = \{s \in Loc_A \mid |s| \in \sigma\}$, where $|s|$ is the underlying justified sequence of the control sequence s. (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 $A \to B$.

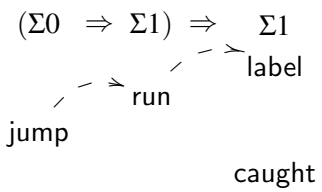
We also observe that $\widehat{\_}$ is functorial on well-bracketed strategies: there is a faithful, identity-on-objects functor $J : \mathscr{G} \to \mathscr{CG}$ sending $\sigma : A \to B$ to $\widehat{\sigma}$.

$\mathscr{CG}$ has structure with which to model $\mathscr{L}$ — (Cartesian closure, strong lifted sum monad $\Sigma_{\_}$ on $\mathsf{Fam}(\mathscr{CG})$, and the functor $J : \mathscr{G} \to \mathscr{CG}$ preserves the meaning of $\mathscr{L}$-terms, so interpreting `new` as the decorated strategy $\widehat{\mathsf{cell}}$ yields a sound and adequate interpretation of $\mathscr{L}_{\mathscr{R}}$.

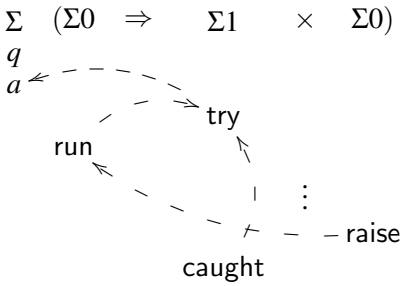
4.1 Denotational Semantics of Continuations and Exceptions

We interpret $\mathscr{L}_{\mathscr{RCE}}$ in $\mathsf{Fam}(\mathscr{CG})$ by extending the semantics of $\mathscr{L}$ with denotations for the constants `callcc` and `new_exn` defined directly as (non-well-bracketed) strategies.

Continuations We interpret `callcc` by decorating the strategy $\mathsf{callcc}_{A,B} : ((A \Rightarrow \Sigma B) \Rightarrow \Sigma A) \to \Sigma A$ defined in [7] with control pointers. This responds to the initial question (label) with a Player question (run), and to its answer or the subsequent question (jump) with an answer to the initial question (caught) (note that this violates the bracketing condition), and thereafter plays copycat between moves hereditarily enabled by (run), and those hereditarily enabled by caught. Control pointers from Player questions point to the pending question, and from Opponent questions, may point to any question. Thus a typical play of $\mathsf{callcc}_{1,0}$ is as follows:



Exceptions The new-exception strategy $\mathsf{exn}_A : \Sigma((\Sigma A \Rightarrow \Sigma 1) \times (\Sigma A))$ is as defined in [8]: it relies on control pointers to determine the current exception handler. Here is a typical play:



Its behaviour can be described informally as follows:

- Answer the initial question.
- If Opponent plays a try move then respond with a 'run' move.
- If Opponent plays a throw move and some try moves are *open*, then answer the most recent one with caught. (Otherwise do nothing, representing divergence caused by an uncaught exception.)

Although these definitions for the denotations for `callcc` and `new_exn` are rather informal, we will show that they may be recovered from more precise continuation-passing and exception-passing interpretations.

5 Composing Continuations and Exceptions

In this section, we construct a semantics of $\mathscr{L}_{\mathscr{R}\mathscr{C}\mathscr{E}}$ by composing exceptions and continuation passing-style interpretations. 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. We first give an "exception-passing" interpretation of $\mathscr{L}_{\mathscr{R}\mathscr{C}\mathscr{E}}$ in $\mathscr{L}_{\mathscr{R}\mathscr{C}}$, followed by CPS interpretation of $\mathscr{L}_{\mathscr{R}\mathscr{C}}$ in $\mathscr{L}_{\mathscr{R}}$. (This decomposition corresponds to the fact that the (strong) exceptions monad $\_+E$ distributes over any other monad, in particular, the continuations monad $\mathscr{R}^{\mathscr{R}^-}$ for any "answer object $\mathscr{R}$, giving an exceptions-and-continuations monad $\mathscr{R}^{\mathscr{R}^{-+E}}$.)

5.1 Exception-Passing-Style Interpretation

We may represent the action of the (global) exceptions monad $\_+E$ as a translation from $\mathscr{L}_{\mathscr{R}\mathscr{C}\mathscr{E}}$ into $\mathscr{L}_{\mathscr{R}\mathscr{C}}$ (and hence by composition with CPS translation into $\mathscr{L}_{\mathscr{R}}$ itself). First, we need to choose an appropriate object E. Since our exceptions do not carry explicit values, we take this to be 1 — the minimal object giving a non-trivial sum. This yields an *exception-passing-style* translation $(\_)^E$ from $\mathscr{L}_{\mathscr{R}\mathscr{C}\mathscr{E}}$ to $\mathscr{L}_{\mathscr{R}\mathscr{C}}$ acting on types as follows:

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

Translation of computations $x_1 : S_1, \ldots, x_n : S_n \vdash M : T$ as computations $x_1 : S_1^E, \ldots, x_n : S_n^E \vdash M^E : T^E + 1$ and values $x_1 : S_1, \ldots, x_n : S_n \vdash V : T$ as values $x_1 : S_1^E, \ldots, x_n : S_n^E \vdash V^E : T^E$ is given in Table 4. The rules for translating the constructs of $\mathscr{L}_{\mathscr{R}\mathscr{C}}$ arise 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:

- $\mathsf{raise}(e) = \lambda z.e := \mathtt{tt}; \mathtt{in}_1(()) : (1 \to 0)^E$

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

Table 3: Exception-passing translation

- $\mathbf{handle}(e) = \lambda f.(f\,());\mathtt{If}\,\mathtt{deref}(e)\,\mathtt{then}\,(e := \mathbf{ff}; \mathtt{in_l}(())))\,\mathtt{else}\,\mathtt{in_r}(()) : ((1 \to 0) \to 1)^E$

New-exception declaration simply aggregates these methods and hides the internal state e.

To establish that the interpretation of local exceptions via translation into $\mathscr{L}_{\mathscr{R}\mathscr{C}}$ is sound, we break the exception-propagation rule down to propagate exceptions past each non-matching handler and show:

Lemma 5.1 *$M \Downarrow$ if and only if $M^E \Downarrow$.*

5.2 Continuation-Passing-Style Interpretation

We now give an interpretation of $\mathscr{L}_{\mathscr{R}\mathscr{C}}$ in $\mathscr{L}_{\mathscr{R}}$ — a quite standard 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 CPS translation is sound with respect to the operational semantics — i.e.

Proposition 5.2 *For any program $M : 1$, $M \Downarrow$ if and only if $M^C\,\kappa \longrightarrow \kappa\,()$*

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

Table 4: CPS translation of exception-free terms

6 Relating the Direct and Indirect Models

In this section, we analyse the relationship between control games and the exception and continuation passing exceptions, with the aim of proving soundness for the former.

First, we recall the observation in [7] that relaxing the bracketing condition on strategies renders the lifted sum of games equivalent to a CPS construction. More precisely, let $\mathscr{G}^C$ be the category in which objects are arenas and morphisms from A to B are unbracketed strategies on $A \Rightarrow B$. Let $U_C : \mathsf{Fam}(\mathscr{G}^C) \to \mathsf{Fam}(\mathscr{G})$ 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 6.1 $\Sigma \cdot U_C \cong U_C \cdot (\Sigma 0)^{(\Sigma 0)^-}$

Using this fact, we describe an intermediate model for $\mathscr{L}_{\mathscr{RCE}}$ based on a category of *exception arenas* in which raising and propagating of the global exception is represented by playing explicit "exception moves". We'll show that this is equivalent to the semantics based on the exception monad, and that it may be related to the control games interpretation by replacing runs of exception moves with control pointers.

An exception arena is a tuple $(M_A, \vdash_A, \lambda_A, e_A)$ consisting 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.

The category of exception arenas, $\mathscr{G}^E$ has exception-arenas as objects and unbracketed strategies on $A \Rightarrow B$ as morphisms from A to B: since the disjoint union and grafting of exception arenas is an exception arena, $\mathscr{G}^E$ is Cartesian closed as for $\mathscr{G}$. There is a full and faithful (identity on morphisms) functor $U_E : \mathscr{G}^E \to \mathscr{G}$ forgetting the exception answer labelling on arenas.

Given a family of exception-arenas B, define $\Sigma_E B$ to be the exception arena given by extending ΣB with an additional answer move a to the initial question, and setting $e(q) = a$. By definition, we have:

Lemma 6.2 *There is an isomorphism of arenas* — $U_E(\Sigma_E B) \cong \Sigma(U_E(B) + 1)$.

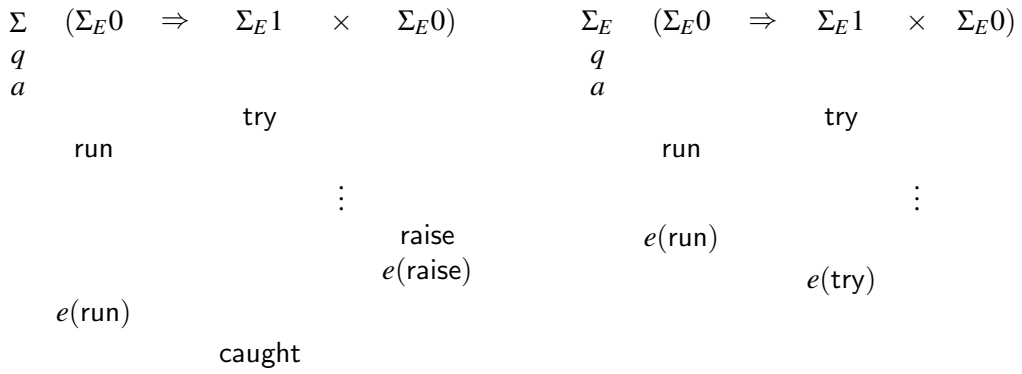


Figure 1: Plays of exn_E

By full faithfulness of U_E, we therefore have a strong monad Σ_E on $\mathsf{Fam}(\mathscr{G}^E)$ such that $U_E \cdot \Sigma_E = \Sigma \cdot U_E$. By Lemma 6.1, Σ_E is equivalent to the exceptions-with-continuations monad $\mathscr{R}^{\mathscr{R}^{-}+1}$, where $\mathscr{R}$ is the one-move game — i.e.

Proposition 6.3 $U_C \cdot U_E \cdot \Sigma_E \cong \mathscr{R}^{\mathscr{R}^{-}+1} \cdot U_C \cdot U_E$.

Moreover, this correspondence extends (up to isomorphism) to the interpretation of $\mathscr{L}$-types.

Proposition 6.4 *For any $\mathscr{L}$-type T, there is an isomorphism of arenas $\phi_T : U_C(U_E([\![T]\!])) \cong [\![T^{EC}]\!]$.*

PROOF: The key type-constructor is the function type: we have $U_E(U_C([\![S \to T]\!])) = U_C(U_E([\![S]\!])) \Rightarrow U_C(U_E(\Sigma_E[\![T]\!])) \cong [\![S^{EC}]\!] \Rightarrow \mathscr{R}^{\mathscr{R}^{[\![T^{EC}]\!]+1}} = ([\![S^{EC}]\!] \times [\![(T^{EC}+1)]\!]) \Rightarrow \mathscr{R}) \Rightarrow \mathscr{R} \cong [\![(S^{EC} \times ((T^{EC}+1) \to 0)) \to 0]\!] \cong (S \to T)^{EC}$. □

Hence we may give a semantics of $\mathscr{L}$ in the Kleisli category of Σ_E on $\mathsf{Fam}(\mathscr{G}^E)$, and interpret computational effects in this model by recovering the images of the continuation/exception passing interpretations of `new`, `new_exn`, `callcc` through this isomorphism.

Take, for example, the interpretation of new exception declaration $\texttt{new\_exn} : ((1 \to 0) \to 1) \times (1 \to 0)$. We have evident raise and handlestrategies which raise 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 exception-declaration strategy $\mathsf{exn}_E : ((\Sigma_E 0 \Rightarrow \Sigma_E 1) \times (1 \to \Sigma_E 0)$ 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. Example plays are given in Figure 1.

6.1 Relating exception-moves and control pointers

We now extend our soundness result to the control-games interpretation of $\mathscr{L}_{\mathscr{RCE}}$ by establishing a correspondence with the exception-arena model: a meaning-preserving functor into $\mathscr{CG}$ from a subcategory of $\mathscr{G}^E$ consisting of exception-arenas and "exception-propagating strategies".

Given an exception-arena A, let $K(A)$ be the arena obtained by erasing all of the exception-answers in A — i.e. $M_{K(A)} = M_A - \{e(q) \mid \lambda(q) = Q\}$. 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 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. 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 (if any).
- 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 exn_E new-exception strategy (Fig. 1) is transformed to the typical play given for the corresponding control strategy.

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 $K(\sigma) = \{K(s) \mid s\sigma\}$ 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 6.5 *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.*
- *$K(s){\restriction}A,C = K(s{\restriction}A,C)$.*

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

Proposition 6.6 *The composition of exception-propagating strategies is exception-propagating, and thus:*

- *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

Lemma 6.7 $\Sigma_E \cdot K = \Sigma \cdot K$.

So for $\mathscr{L}_{\mathscr{RCE}}$-types we have $K([\![T]\!]_E) = [\![T]\!]_C$.

Proposition 6.8 *Every $\mathscr{L}_{\mathscr{RCE}}$-term M denotes an exception-propagating strategy such that $K([\![M]\!]_E) = [\![M]\!]_C$.*

PROOF: For strategies denoting terms of $\mathscr{L}_{\mathscr{RC}}$ (which never raise or handle an exception) this is straightforward. It suffices to verify (by inspection of plays) that exn_E is exception-propgating, and that K sends it to the corresponding control strategy. □

Hence we have shown that:

Proposition 6.9 *Interpretation of programs as control strategies is sound: $M \Downarrow$ if and only if $[\![M]\!]_C \neq \bot$.*

7 Full Abstraction

Finally, we prove full abstraction for the control games model via reduction (by factorization) to the definability result for the original model of $\mathscr{L}_{\mathscr{R}}$. 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.

Proposition 7.1 *[[1]]For any type $\mathscr{L}$-type T, every compact basic strategy on $\Sigma[\![T]\!]$ is the denotation of a $\mathscr{L}_{\mathscr{R}}$ term $M_\sigma : T$.*

Since $J : \mathscr{G} \to \mathscr{CG}$ 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 constraint:

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

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

Lemma 7.3 *For any (finitary) 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 defined in [6]: 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 $\mathsf{label}, \mathsf{run}\ \mathsf{jump}, \mathsf{caught}$ move-pairs between each Opponent and Player moves in s, so that any intervening questions are closed. $s_1 \sqcap s_2$ is even-length then so is $\widetilde{s_1} \sqcap \widetilde{s_2}$. Thus if $t_1, t_2 \in \widetilde{\sigma}$ then there exists $s_1, s_2 \in \sigma$ with $t_1 \sqsubseteq \widetilde{s_1}, t_2 \sqsubseteq \widetilde{s_2}$, and $s_1 \sqcap s_2$ is even-length, which implies $t_1 \sqcap t_2$ is even-length.

□

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 control strategy $\sigma : 1 \to A$ there is a 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 interjects a raised and then a handled exception between Opponent and Player moves. The handler which catches the raised exception is determined by the control pointers on s, so that $|\widehat{s}| = |\widehat{t}|$ implies $s = t$. Since $\widehat{s}{\restriction}A = s$, and $\widehat{s}{\restriction}[\![\mathtt{exn}]\!] \in \mathsf{exn}$, 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 [10] 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" [13]. 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*.

[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.

[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.

[4] K. Honda & N. Yoshida (1997): *Game theoretic analysis of call-by-value computation*. In: *Proceedings of ICALP '97*, *Lecture Notes in Computer Science* 1256, Springer-Verlag.

[5] 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.

[6] 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.

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

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

[9] J. Laird (2002): *Exceptions, Control and Macro-expressiveness*. In: *Proceedings of ESOP '02*, *LNCS* 2305, Springer.

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

[11] 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.

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

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