Software Fault-Tolerance in Functional Programming *

R. Govindarajan
Dept. of Electrical Engineering, McGill University, Montreal, H3A 2A7, Canada
govindr@pike.ee.mcgill.ca

Abstract

Handling exceptions in functional languages has always been a problem, due to the inherent conflict between the control flow oriented approach of exception handlers and the functional style of evaluation. In this paper, we propose a new approach that discards the conventional view of treating exceptions as a means of effecting control transfer. Instead, exceptions are used to change the state of an object: A Terminate exception is viewed as 'shielding' the input object. On the other hand, a Resume exception designates the input object as 'curable' and requires the immediate application of a handler function. This approach enables the description of the semantics of exception raising functions (i) without associating any implementation restriction and (ii) without the loss of referential transparency and commutativity of functions.

1 Introduction

Functional languages describe algorithms in a clear, concise and natural manner. They are free from side-effects, and express parallelism in a natural way. These properties make functional languages attractive not only from the theoretical perspective, but also from the program construction point of view.

A key issue in program construction is robustness [8]. Software reliability can be achieved by the judicious use of fault-tolerant tools. Exception handling [4] is one of the two techniques used for developing reliable software. Only a few functional languages, namely Standard ML [7], Parallel Standard ML (PSML) [6], ALEX [3], Gerald [9], and Functional Languages (FL) [2], support constructs for software fault-tolerance.

In this paper, we define and develop the required notations for specifying Terminate and Resume exceptions. Though the proposed notations can, in principle, be used for any functional or applicative language, we choose Backus' Functional Programming (FP) languages [1] for expository purposes. Compared to other related works, our approach has the advantage of not enforcing any implementation restriction, such as sequentializing the execution or prioritizing the order of evaluation. In the following section, we discuss the issues in incorporating exception handlers in functional languages. The subsequent two sections presents our approach to embedding exception handlers in FP. Finally, we compare our work with other related works in Section 5.

2 Background

In the following subsection, we present the issues in embedding exception handling constructs in functional programming.

2.1 Related Issues

In imperative languages, an exception is conventionally treated as a means of effecting a control transfer. This poses a fundamental conflict with the functional style of computation. Due to this, exceptions in functional languages can result in non-deterministic behavior of the FP program. This is illustrated by means of the following example by Bretz [3].

\[
\lambda x. x \text{ terminate in}
\]

An application of this expression to any input object could yield 3 or 5 depending on the order in which the subexpressions are evaluated.

This problem has been considered as intrinsic to incorporating exceptions in functional languages. Languages ML [7] and ALEX [3] circumvent this problem by proposing sequential execution. Such a restriction is severe and is essentially required because the control flow view of exceptions has been carried through to functional languages. We discard this view and define the semantics of FP functions operating on exceptional objects without imposing any restriction on the execution. Though Gerald [9] and PSML [6] allow

*This work was supported by MICRONET — Network Centres of Excellence, Canada.
parallel execution of subexpressions, they retain deterministic behavior by assigning priorities to exceptions.

Secondly, Exception handling might cause side effects in expressions and hence might violate the property of referential transparency. Proposed solutions [3, 7] suggest the association of an environment with the functions. But in our case, discarding the conventional control flow view of exception solves this problem naturally. In [5] we establish that the introduction of new constructs does preserve the algebraic properties of FP. Our approach, like PSML [6], uses error values for handling exceptions. However, there are some important differences between the two. Section 5 brings out these differences.

Lastly, embedding exception handling constructs in lazy functional language can transform otherwise non-strict functions into hyper-strict functions [9]. Reeves et al. [9] point out this with the help of the following example.

\[ \text{handle bad by } \lambda x.0 \text{ terminate in } \left((12 + (\text{signal bad } 13)) \oplus 14\right) \]

where \( \oplus \) is any non-strict binary operator, non-strict in both arguments. The signal \( \text{bad} \) propagates up through the operators + and \( \oplus \), to be handled by the respective handler function. Even though \( \oplus \) is defined to be non-strict in both arguments, the up-propagation of signal through \( \oplus \) makes it strict. This is because, both subexpressions need to be evaluated to determine whether or not they raise any exception. Reeves et al. claim the transformation of non-strict actors into hyper-strict is due to the up-propagation of signals through non-strict operators. To overcome this problem, the notion of down-propagation and firewalls has been defined in [9]. In this paper, however, we argue that it is not the up-propagation, but the 'persistent' nature [2] of exception values that causes the above problem. By this we mean if an exception value is a part of an object as \( \langle X_1, \ldots, e, \ldots, X_n \rangle \), then because of its special status, the error signal \( e \) 'persists' and the above object is indistinguishable from \( e \). This is also referred as following strict semantics on exception values. In Section 3.2 we illustrate this with the help of an example and show how the above problem could be overcome by following lazy semantics for error values.

2.2 Our Approach

Terminate Exceptions

If the application of a function \( F \) on an object \( X \) results in a \textit{Terminate} exception \( e \), then the input object is considered to be shielded by \( e \). Such an object is represented as \( X^e \). Except for the respective handler, all functions operating on the shielded object are inhibited. In other words, when an object is shielded, any function applied to it behaves like the identity function. The handler \( H \) for an exception \( e \), when applied to \( X^e \), removes the shield of the object and results in \( H : X \). It must be noted that a handler can remove only the corresponding shield. Thus, if an object is shielded by a number of exceptions, shields can be removed only if the respective handlers are applied in the appropriate order. Further, the domain of the FP system has been extended to include shielded objects. Shielded objects are not 'persistent' [2] and can be elements of composite objects. Also, functions are defined to be non-strict with respect to shielded objects\(^1\). The semantics of FP functions and functional forms operating on shielded objects and partially shielded objects are described in the next section. The reason for choosing non-strict semantics (henceforth in this paper non-strict refers to being lazy with respect to shielded objects) is to allow shielded objects to exist and carry over strict FP functions and functional forms, if any, and ultimately to be handled by the appropriate handler.

Resume Exceptions

\textit{Resume} exceptions are handled as locally as possible and therefore embedding them in functional languages do not cause any fundamental conflict. Since \textit{Resume} exceptions are handled locally, the concept of shielding an object does not help. However, such sophisticated handling is not required for \textit{Resume} exceptions if we view the situation in the following manner. When a \textit{Resume} exception is raised, the input object is considered to have some abnormality which needs to be 'cured' immediately. Instead of passing the object (possibly with a shield) to the handler, the handler is invoked at the activation point as an immediate cure function. Such a view is simple and serves the purpose.

In the following sections, we introduce notations and constructs for \textit{Terminate} and \textit{Resume} exceptions. As the new constructs are defined, the domain and semantics of the FP functions will be redefined as required.

\(^1\)It is equally easy to define the FP functions and functional forms in a non-strict manner with respect to the undefined object, \( \perp \). However, we do not attempt this here.
3 Terminate Exceptions

The objects, functions, and functional forms of FP are extended in the following way to embed Terminate exceptions.

3.1 The Extended FP System

Objects

An object \(X\) shielded by an exception \(e\) is represented as \(X^e\). The exceptional object \(X^e\) can be a component of a composite object \(X_1\). Then, the objects \(X^e\) and \(X_1\) are respectively known as fully and partially shielded objects. In fact, a shielded object \(X_1\) or \(X^e\) can have more than one fully shielded object, possibly shielded by different exception names. Thus, \((X_1, X_1^e, X_1^e, \ldots, X_n^e)\) is a partially shielded object. A fully shielded object cannot be shielded by another exception name. That is, \(X^e\) is invalid. However, \((X_1, X_2, \ldots, X_n)^e\) is valid, and interestingly \(e\) can be same as \(e\).

Formally, an object can be undefined (denoted by \(\bot\)), or an atom, or a sequence of objects of the form \((X_1, \ldots, X_n)\), where each \(X_i\) is an object which could be (i) unshielded, (ii) partially shielded or (iii) completely shielded by a single exception. That is, the domain \(O\) of objects can be divided into three disjoint sets, namely (i) the set \(C\) of completely shielded objects of the form \(X^e\), (ii) the set \(P\) of partially shielded objects of the form \((X_1, X_1^e, \ldots, X_n^e)\) and \(\exists X_i\) such that \(X_i \in C \cup P\), and (iii) the set \(N\) of normal or unshielded objects of the form \((X_1, X_2, \ldots, X_n)\), where \(\forall X_i, X_i \notin C \cup P\). The sets \(N, P,\) and \(C\) are such that \(O = N \cup P \cup C\).

Primitive Functions

The semantics of the primitive functions operating on shielded objects is defined below. The meaning of these functions when applied to normal objects is as in [1].

Selector Functions

\[s : X \equiv X, \quad \text{if } X \in P\]
\[s : X \equiv X, \quad \text{if } X \in C\]

Tail

\[tl : X \equiv (X_2, \ldots, X_n)\]
\[\quad \text{if } (X \in P) \land (X = (X_1, X_2, \ldots, X_n))\]
\[\quad \equiv X, \quad \text{if } X \in C\]

Identity

\[\text{Id} : X \equiv X, \quad \forall X \in P \cup C\]

Atom

\[\text{Atom} : X \equiv \text{False}, \quad \text{if } X \in P\]
\[\quad \equiv X, \quad \text{if } X \in C\]

Equal

\[\text{Equal} : X \equiv X, \quad \forall X \in P \cup C\]

Null

\[\text{Null} : X \equiv \text{False}, \quad \text{if } X \in P\]
\[\quad \equiv X, \quad \text{if } X \in C\]

Reverse

\[\text{Reverse} : X \equiv (X_n, \ldots, X_1)\]
\[\quad \text{if } (X \in P) \land (X = (X_1, \ldots, X_n))\]
\[\quad \equiv X, \quad \text{if } X \in C\]

Distribute from Left

\[\text{Distl} : X \equiv (\{(Y, X_1), \ldots, (Y, X_n)\})\]
\[\quad \text{if } (X \in P) \land (X = (Y, (X_1, \ldots, X_n)))\]
\[\quad \equiv X, \quad \text{if } (X \in P) \land (X = (Y, X_1))\]
\[\quad \equiv X, \quad \text{if } X \in C\]

The function Distribute from right can similarly be defined.

Add, Subtract, Multiply, Divide, And, Or, Not

\[\text{Add} : X \equiv X, \quad \forall X \in P \cup C\]

Functions Sub, Mult, and Div can be similarly defined.

Next we deal with the functional forms.

Functional Forms

Composition

\[(f \circ g) : X \equiv f : (g : X), \quad \forall X \in P \cup C\]

Construction

\[(f_1, \ldots, f_n) : X \equiv (f_1 : X, \ldots, f_n : X), \forall X \in P \cup C\]

Condition

\[(p \rightarrow f \circ g) : X \equiv p : X, \quad \text{if } (p : X) \in P \cup C\]
\[\quad \equiv f : X, \quad \text{if } (p : X) \text{ is True}\]
\[\quad \equiv g : X, \quad \text{if } (p : X) \text{ is False}\]
\[\quad \equiv \bot, \quad \text{otherwise}\]

Constant

The constant functional is non-strict over partially and fully shielded objects. This means,

\[n : X \equiv n, \quad \forall X \in P \cup C\]
Apply to all
\[\alpha f : X \equiv X, \quad \text{if } X \in C \]
\[\equiv \{ f : X_1, \ldots, f : X_n \} \]
\[\text{if } (X \in P) \land (X = (X_1, \ldots, X_n)) \]

Definitions

The definitions of an FP program are written in the form:
\[\text{DEF (func-name) } \equiv \{ \text{ func-defn } \} \]
\[\text{SIGUALS } (\text{exception-list}) \]
where (func-name) and (func-defn) represent the function name and the function body respectively. The term inside the braces represents an optional occurrence. That is, the term is included only if the function raises any exception. The exceptions raised by the function are listed in exception-list.

Programming Exceptions

A Terminate exception can be raised using the Raise function (similar to the signal function in ALEX or Gerald) and an exception name \(e\). That is, 
\[\text{Raise } e : X = X^e, \quad \forall X \in N \cup P \]
\[\text{Raise } e : X = X, \quad \forall X \in C \]
Adding syntactic sugar, the Raise function can be written in FP style as \(\text{Raise } \circ [e, \text{Id}]\). However, we continue to use the representation \(\text{Raise } e\) for simplicity sake.

Handler Functions

A handler for a Terminate exception \(e\) is written as \(\lbrack e \Rightarrow H \rbrack\) (read as 'on \(e\) do \(H\)'). The application of this function, called the handler function, on \(X\) is defined as:
\[\lbrack e \Rightarrow H \rbrack : X \equiv H : X_1, \text{if } (X \in C) \land (X = X_1) \]
\[\equiv X, \quad \text{otherwise.} \]
In order to write handler functions in FP style, a function \(\text{Hand}\) can be defined as
\[\lbrack e \Rightarrow H \rbrack \equiv \text{Hand } \circ [e, H, \text{Id}]\].
However we prefer the \(\lbrack e \Rightarrow H \rbrack\) notation. Further, from the definition of Raise and handler functions, it can be observed \(\forall X \in N \cup P, \) and for any \(f,\)
\[f : X = \lbrack e \Rightarrow f \rbrack : X^e, \]
where \(e\) is any exception name.
Finally, the default handler can be written as \(\lbrack ? \Rightarrow H \rbrack\) and has the following semantics.
\[\lbrack ? \Rightarrow H \rbrack : X \equiv H : X_1, \quad \text{if } (X = X_1) \]
\[\equiv X, \quad \text{otherwise.} \]

We illustrate the notations introduced in this section by an example.

Example 3.1

The ADD-SUBEXP problem discussed in Section 2.1. The problem can be programmed in FP as:
\[\text{DEF Exp1 } \equiv \text{True } \rightarrow \text{Raise } e_1 \]
\[\text{DEF Exp2 } \equiv \text{True } \rightarrow \text{Raise } e_2 \]
\[\text{DEF Add-Subexp } \equiv \text{Add } \circ [\text{Exp1}, \text{Exp2}] \]
\[\text{SIGUALS } e_1, e_2 \]
\[\text{DEF Robust-Add-Subexp } \equiv \text{Add } \circ \{ e_1 \Rightarrow 3 \} \circ \{ e_2 \Rightarrow 5 \} \circ \text{Add-Subexp} \]

The application of (ADD-SUBEXP : 1) results in \((1^e, 1^e)\). Applying \(\text{Add } \circ \{ e_1 \Rightarrow 3 \} \) to \((1^e, 1^e)\), we get \((1^e, 5)\). Finally \(\text{Add } \circ \{ e_1 \Rightarrow 3 \} : (1^e, 5)\) yields Add : \((3, 5)\) = 8. Observe that the result would be the same irrespective of the order in which the subexpressions EXP1 and EXP2 are evaluated.

3.2 Remarks

(1) In our proposal, the user has the freedom to define the handler anywhere he likes. Shielded objects (partially or fully) propagate through strict functions and reach the handler. The propagation is implicit. However, non-strict functions may have the dangerous effect of (partially or completely) pruning the exceptional object. Hence, care need to be exercised in placing the handler functions.

(2) A shielded object propagates through the dynamic invocation chain until it encounters an appropriate handler. Hence, the handler association is 'dynamic'. It may be observed that our view of Terminate exceptions (that they shield abnormal values) facilitates dynamic handler association in a natural manner.

(3) Consider the following function
\[\text{Circled-plus } \equiv 2 \circ [+ \circ [1, (\text{Esc } e_1) \circ 2], 3] \]
\[\text{SIGUALS } e_1. \]

The expression \(\lbrack e_1 \Rightarrow 0 \rbrack\) o Circled-plus : \((12, 13, 14)\) is similar to
Circled-plus \(\equiv 2 \circ [\!\! + \!\! \circ [1, (\text{Raise } e1) \circ 2] \!\!] 3\)

where the select function is used in the former in the place of the operator \(\oplus\). In [9] it has been argued that the up-propagation of the signal bad transforms \(\oplus\) into a strict function. However, reducing Circled-plus: \((12,13,14)\) yields \(2 : \{(12,13\varepsilon), 14\}\) which in turn returns the value 14. Thus, the evaluation remains non-strict even in the presence of the exception \(e1\) and its up-propagation. However, if we allow exception objects to be persistent, the application of Circled-plus on \((12,13,14)\) would have resulted in \(2 : (13\varepsilon)\), making the select function strict. Thus, we conclude that it is essentially the persistent nature of exception objects that transforms non-strict functions into strict functions.

4 Resume Exceptions

Resume exceptions are denoted by \(\varepsilon\), where \(e\) could possibly be subscripted. The function \(\text{Res } \varepsilon\) is introduced to raise the Resume exception. The handler \(H\) for a Resume exception is represented as \(\llbracket \varepsilon \Rightarrow H \rrbracket\), the influx symbol over \(\varepsilon\) indicating that \(\varepsilon\) is of type Resume. As mentioned earlier, when the application of a function on an object \(X\) raises a Resume exception, the object \(X\) is treated as curable; the corresponding handler function is applied to the object \(X\) at the activation point. A runtime error occurs when a Res \(\varepsilon\) function is invoked without a corresponding handler or default handling function.

Resume exceptions do not shield objects, and the existing definitions of objects, functions and functional forms are sufficient to express them. Explicit compile time techniques are required to accomplish dynamic association of handler to a Resume exception. It is beyond the scope of this paper to discuss an implementation for the association of handlers.

Next, we present a simple example to illustrate how Resume exceptions can be programmed.

Example 5.1

This example deals with a program that adds the magnitudes of a sequence of numbers. The program calls a function SUM which adds two numbers. Exceptions are raised in SUM whenever one or both of the numbers considered for addition are negative. The handler takes an input argument and returns the magnitude of it. The SUM operation is resumed after handling the exception.

\[\text{DEF COND } \equiv (Lt \circ [Id, 0] \rightarrow \text{Res } \varepsilon; Id)\]

\[\text{DEF SUM } \equiv \llbracket \varepsilon \Rightarrow \text{Negate} \rrbracket \circ \text{Addo} \]

\[\text{DEF CUM-SUM } \equiv / \text{SUM}\]

The Negate function negates the input argument. It can be seen that the COND function in the above example raises \(\varepsilon\) whenever its input is negative. The cure function Negate is immediately applied to the input object when the exception \(\varepsilon\) is raised.

At first sight, functions raising Resume exceptions appear to violate referential transparency. Consider a function \(F\) which raises a Resume exception \(\varepsilon\). Suppose \(H\) is the handler for \(\varepsilon\), not necessarily associated with \(F\). Now \(F.X\) may result in one of the two values, depending on whether the object lies in the normal or exceptional domain of \(F\). Thus it appears that \(F\) is non-deterministic. But this is not so and can be reasoned in the following way. The function \(F\) can be considered as a (sort of) higher order function with \(H\) as an argument. The function \(F\) selectively applies \(H\) whenever the input object is in the exceptional domain. That is, \(F\) is a deterministic function whose range can be divided into two sets, one corresponding to the normal values and the other for exceptional values.

5 Related Work

There have been a few attempts [2, 3, 6, 7, 9] to incorporate exception handling in functional or applicative languages. We compare the earlier proposals with ours.

(1) The main difference between our work and the related ones is the radical change in the way exceptions are viewed. Earlier works [3, 7] treat exceptions as a means of effecting a control transfer. This leads to a fundamental conflict and as a remedy requires the imposition of sequential execution. We treat exceptional objects either as shielded or as requiring immediate application of cure functions. This approach naturally suits the functional style and, therefore, does not necessitate any constraint on execution.

(2) Our approach is similar to PSML in that both use error data values to handle exceptions. However in PSML deterministic program behavior is regained by assigning priorities to exceptions. Besides, prioritizing exceptions involves additional implementation overheads. Gerald, which uses the replacement model of Yemini and Berry [10], also prioritizes exceptions to guarantee deterministic behavior. On the other hand our approach, by following non-strict semantics, avoids the need for such unnecessary requirements.
In our scheme, an exception is automatically propagated to higher level modules until an appropriate handler is found. That is, the propagation of an exception along the dynamic invocation chain is transparent to the user. The programming language ML [7] also supports implicit propagation of exceptions; in contrast, in ALEX [3], the exceptions must be explicitly transmitted.

ML [7] supports only \textit{Terminate} type of exceptions; whereas our work, like ALEX, allows both \textit{Resume} and \textit{Terminate} exceptions.

FL language [2] has been designed with constructs for exception handling. Though user-defined exceptions can be programmed in FL, exceptions are mainly used to signal application of inappropriate input object to primitive functions. Also, only one type of exceptions is allowed in FL. Further, the semantics of \textit{Catch} function (equivalent to the handler) is too primitive and may result in deeply nested structures if association points were chosen far away (in the dynamic execution chain) from the activation point. Compared to this, our proposal has a more versatile handler, and the association of handler far away from the activation point does not lead to nested structures.

\section{Conclusions}

In this paper we have introduced the notations for exception handling in FP languages. The notations introduced are clearly illustrated with the help of example programs. In incorporating exceptional handling in FP, the conventional view of treating exceptions as a means of effecting control transfer has been discarded. We treat exceptional objects either as shielded or as requiring immediate application of cure functions. This allows us to describe the exceptions in a functional framework, retaining referential transparency and the nice mathematical properties of functional languages. In fact, embedding exception handling construct in FP has been accomplished without imposing additional execution constraints, such as sequentializing the execution or prioritizing the exceptions. Further, the semantics of the primitive functions of FP are defined in a non-strict manner over the exceptional objects and does not introduce \textit{hyper-strictness}.

\section*{Acknowledgments}

The work presented in this paper would not have taken its present shape without the numerous discussions the author had with Prof. R.A. Nicholl, University of Western Ontario, London, Canada.

\section*{References}


