Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Alexander Konstantinov

Flexible Ausnahme- und Fehlerbehandlung mittels eines Conditionsystems in Scala

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexander Konstantinov

Flexible Ausnahme- und Fehlerbehandlung mittels eines Conditionsystems in Scala

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Michael Böhm
Zweitgutachter: Prof. Dr. Thomas Thiel-Clemen

Eingereicht am: 6. Juli 2011

Alexander Konstantinov

Thema der Arbeit

Flexible Ausnahme- und Fehlerbehandlung mittels eines Conditionsystems in Scala

Stichworte

Programmiersprachen, Scala, Fehlerbehandlung, Exceptionsysteme, Conditionsysteme, Common Lisp, Smalltalk, Scheme, Continuations, statische Typisierung, Introspektion, Abstraktion

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Fehlerbehandlung in Programmen, mit einem besonderen Augenmerk auf Fehlerbehandlungssysteme welche von der jeweiligen Programmiersprache bereitgestellt werden. Es werden verschiedene Möglichkeiten der Fehlerbehandlung auf ihre Funktionsweise und ihre Nützlichkeit zur Behandlung von Ausnahme- und Fehlersituationen untersucht.

Ein wichtiger Teil der Arbeit befasst sich mit der Beschreibung und Untersuchung der Fehlerbehandlungssysteme der Programmiersprachen Common Lisp und Smalltalk.

Das dabei gewonnene Wissen wird dann zum Entwurf und der Implementierung eines neuartigen Fehlerbehandlungssystems in Scala genutzt.

Alexander Konstantinov

Title of the paper

Flexible error and exception handling using a condition system written in Scala

Keywords

programming languages, Scala, error handling, exception systems, condition systems, Common Lisp, Smalltalk, Scheme, continuations, static typing, reflection, abstraction

Abstract

This thesis deals with error handling in computer programs and focuses specifically on error-handling systems that are integrated into programming languages. The different means of handling errors viewed in terms of mechanics and their usefulness for dealing with errors and exceptional situations.

An important part is the description and analysis of Common Lisp's and Smalltalk's exception systems.

The rest is a repackaging of those concepts, inspirations and acquired knowledge into a design and implementation of a novel error handling system using Scala programming language.

Inhaltsverzeichnis

1 Einführung **1**

2 Prozedurale Abstraktion **2**
2.1 Polymorphie und Erweiterbarkeit 4
2.2 Anforderungen an Fehlerbehandlungssysteme 4

3 Fehlersignalisierung **6**
3.1 Signalisierungspattern 8
3.1.1 Spezielle Rückgabewerte 8
3.1.2 Rückgabe einer Nullreferenz 9
3.1.3 Nullobjekt-Pattern 10
3.1.4 Multiple Rückgabewerte 13
3.1.5 Call-by-reference- und `out`-Parameter 15
3.1.6 Globale Flags 17
3.1.7 Optiontypes 18
3.1.8 Fazit 20
3.2 Nicht-lokale Fehlersignalisierung 21
3.2.1 Errorhandler 22
3.2.2 Explizite Fehlercontinuations 25
3.2.3 Implizite Fehlercontinuations 28
3.2.4 Fazit 30
3.3 Güte von Signalisierungsmethoden 31

4 Fehlerklassifizierung **32**
4.1 Keine Klassifizierung 32
4.2 Errorcodes 33
4.3 Symbolische Klassifizierung 34
4.4 Fehlerhierarchien 35
4.4.1 Klassifikation mittels Mehrfachvererbung 38
4.4.2 Reinterpretation von Fehlern 38
4.5 Fazit 42

5 Fehlerbehandlung **43**
5.1 Unzulänglichkeiten von `throw-catch`-Exceptionsystemen 43
5.1.1 Explizite Fehlerbehandlungsstrategien 46
5.1.2 Alternativsemantik für `throw-catch` 49

6 Analyse der Fehlerbehandlungssysteme von Smalltalk und Common Lisp 53
6.1 Das Conditionsystem von Common Lisp . . . 53
6.1.1 Fehlersignalisierung . . . 54
6.1.2 Fehlerklassifizierung . . . 57
6.1.3 Fehlerbehandlung . . . 58
6.1.4 Fazit . . . 63
6.2 Das Exceptionsystem in Smalltalk . . . 63
6.2.1 Fehlersignalisierung . . . 64
6.2.2 Fehlerbehandlung . . . 67
6.2.3 Fehlerklassifizierung . . . 70
6.2.4 Fazit . . . 71
6.3 Konzeptionelle Unterschiede . . . 72

7 Design und Implementierung eines Conditionsystems für Scala 73
7.1 Designziele . . . 73
7.2 Schnittstellen . . . 77
7.2.1 Handlerdeklaration . . . 77
7.2.2 Fehlercontinuations . . . 78
7.2.3 Signalisierungskombinationen . . . 80
7.2.4 Introspektion von Fehlercontinuations . . . 80
7.2.5 Introspektion von Handlern . . . 82
7.2.6 Hookmethoden in der Conditionhierarchie . . . 83
7.3 Anwendungsfälle . . . 84

8 Fazit 87

1 Einführung

Bei der Programmierung von Software ist Robustheit, also der Erhalt von Funktionalität auch bei unerwarteteren Ereignissen und Abnormitäten, ein sehr erwünschtes Merkmal. Um dieses zu erreichen wurden vielfach Werkzeuge und Techniken zur Ausnahmebehandlung entwickelt, welche in der modernen Softwareentwicklung unterschiedliche Verbreitung fanden. Zu den ausgefeiltesten Werkzeugen gehören die Condition- und Exceptionsysteme von Lisp- und Smalltalk-Dialekten, welche speziell auf die Bedürfnisse von flexibler und dynamischer Software ausgelegt sind.

Diese Arbeit versucht diese Condition- und Exceptionsysteme und andere Werkzeuge und Techniken zur Ausnahmebehandlung in einzelne Konzepte zu zerlegen, diese Konzepte zu erläutern und anschließend Designentscheidungen für eine Implementation eines neuen Conditionsystems zu begründen.

Dazu wird im Verlauf der Analyse bestehender Werkzeuge zur Fehlerbehandlung, ein Prototyp Fehlersignalisierungs- und Fehlerbehandlungssystem in der Programmiersprache Scheme entwickelt, welches auch benutzt wird um Schwierigkeiten bei der Benutzung und Implementation von Fehlerbehandlungssystemen im Allgemeinen aufzuzeigen.

2 Prozedurale Abstraktion

Die Programmiersprachen, die heutzutage Verwendung finden, verfügen alle über eine Form der Prozedur als Abstraktionsmittel, wobei sie dafür, abhängig vom Programmierparadigma, unterschiedliche Namen haben. Ob sie Prozeduren oder Subroutinen in der prozeduralen, Methoden in der objektorientierten oder Funktionen in der funktionalen Programmierung heißen, sie haben gemein, dass sie Abstraktionen bilden. Der Prozeduraufrufer parametrisiert die Prozedur, übergibt den Kontrollfluss an sie und erwartet ein Resultat (einen Rückgabewert und/oder Seiteneffekt), wobei der Aufrufer nichts über die Implementierung der Prozedur wissen muss.

Prozeduren können selber natürlich weitere Prozeduren aufrufen, welche wiederum Prozeduren aufrufen können. Dies führt dazu, dass sich der Kontrollfluss eines Programms einen Baum bildet.

Komplexe Probleme werden von Prozeduren dadurch erledigt, dass sie in einfachere Teile zerlegt werden und diese Teile an Unterprozeduren weiterdelegiert werden, welche diese wiederum zerlegen und weiterdelegieren, bis die Probleme von den Primitiven der Programmiersprache erledigt werden können. Dies erlaubt eine Kategorisierung in “low-level”- und “high-level”-Prozeduren, wobei “high-level”-Prozeduren komplexe Probleme durch das Weiterdelegieren an “low-level”-Prozeduren erledigen. Rekursion ausgenommen, stellt der Aufruf einer Prozedur den Eintritt in eine andere Abstraktionsschicht dar, wobei man sich einen Schritt von dem Ursprünglichem Problem entfernt, und ein einfacheres Problem löst.

Eine weiterer Sinn des Prozedurkonzepts (neben der Aufteilung einer Problemstellung) ist die Wiederverwertbarkeit: Anstatt ähnlichen Code an mehreren Stellen zu haben, wird dieser in eine Subprozedur ausgelagert und diese mehrmals aufgerufen. Eine wiederverwendbare Prozedur kann also nicht wissen, welche Prozedur sie aufgerufen hat, und wieso sie aufgerufen wurde. Aus dem folgt eine n:1-Beziehung zwischen Aufruferprozeduren und einer aufgerufenen Prozedur, wobei die Menge der möglichen Aufruferprozeduren unbeschränkt und erweiterbar ist.

Was dies zur Folge hat, ist dass Code höherer Abstraktionsschichten leichter refaktorisiert und dessen Schnittstellen leichter verändert werden können, da wahrscheinlich weniger Code direkt oder indirekt von ihm abhängt.

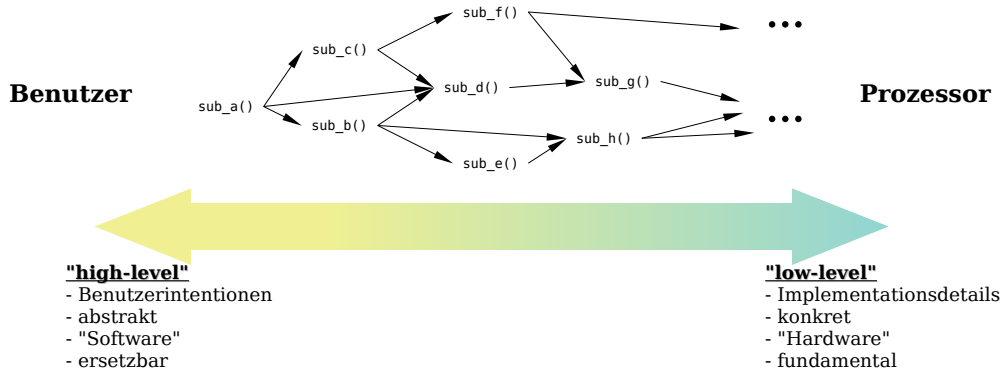

Abbildung 2.1: Abstraktion in Software

Es lässt sich also sagen, dass mit jedem Subprozeduraufruf (rekursive Aufrufe ausgenommen) das Abstraktionsniveau sinkt, die das Wissen über die ursprüngliche Problemstellung abnimmt, der Code vielseitiger nutzbar wird und mehr auf die konkrete Implementierung eingegangen wird.

Das Prozedurmodell erreicht zwar Wiederverwendung und Abstraktion, stößt jedoch auf Probleme bei Ausnahme- und Fehlersituationen: Eine “low-level”-Prozedur hat die Möglichkeit eine Fehlersituation zu erkennen, aber, da sie von verschiedenen Orten heraus aufgerufen werden kann, selten die Möglichkeit diese Fehlersituation zu interpretieren und darauf zu reagieren. In höheren Abstraktionsschichten hingegen kann die Bedeutung von Fehlern besser eingeschätzt werden, da dort eventuell noch Information zur Intention des Nutzers vorhanden ist.

2.1 Polymorphie und Erweiterbarkeit

Wenn Prozeduren nur über ihre abstrakten Schnittstellen angesprochen werden, dann ist es, durch das Ersetzen einer Implementation durch andere, möglich, die Funktionalität von bestehendem Code zu verändern und erweitern.

Zum Beispiel könnte eine Implementation einer Datei-API mit einer `open(Path p)` Prozedur durch eine Ersetzt werden, die nicht nur auf lokale Dateien, sondern auch auf Dateien auf Webservern zugreift. Klienten dieser API hätten dann, ohne ihren Code anzupassen, erweiterte Funktionalität.

Besonders in objektorientierten Programmiersprachen ist der Austausch von Implementationen, um immer im Nachhinein Software noch erweitern zu können, leicht möglich. Dadurch wird aus einer 'm:1'-Beziehung zwischen Aufrufern und Implementation eine 'm:n'-Beziehung, da der Aufrufer sich nicht sicher sein kann, welche Implementation er aufruft. Also ist es nicht nur so, dass er nichts über die konkrete Implementation wissen darf, sondern dass er nichts über die Implementation wissen kann.

Diese Polymorphie hat daher auch Auswirkungen bei Fehlersituationen: Ob und welche Fehler auftreten können hängt immer von der konkreten Implementation ab.[1] Doch auf die Implementation darf der Aufrufer eigentlich keine Einsicht haben.

Desweiteren gibt es dadurch eigentlich keinen Prozeduraufruf mehr, welcher garantiert nicht fehlschlagen kann. Schließlich ist jede Operation potentiell als ein entfernter Aufruf und jede Datenstruktur als ein transparenter Zugriff auf eine Datenbank implementierbar.

2.2 Anforderungen an Fehlerbehandlungssysteme

Zusammengefasst wirken sich Ausnahmesituationen negativ auf die Abstraktionsfähigkeiten von Prozeduren aus, müssen aber auf verschiedenen Wegen behandelt werden, um die Robustheit von Programmen zu gewährleisten. Dabei müssen verschiedene Abstraktionsschichten miteinander kommunizieren ohne unnötige Abhängigkeiten zwischen ihnen zu schaffen.

[1] Ein Beispiel wäre das Laden von einer Konfigurationsoption: Eine Abfrage über ein Dialogfenster kann Fehler aus dem GUI-Toolkit und das Auslesen einer Konfigurationsdatei Fehler vom Dateisystem auslösen

Wenn die Aufgabe von Programmiersprachen darin besteht, Abstraktionsmittel bereitzustellen[HAS96, S. 17] und prozedurale Abstraktion in der Anwesenheit von Fehlersituationen versagen kann, muss eine Programmiersprache auch Mittel und Systeme anbieten um mit ihnen umzugehen.

Diese Systeme zur Fehler- und Ausnahmebehandlung sollten

- möglichst wenig umständliche Propagation von Fehlern an höhere Abstraktionsschichten erlauben, da sonnst die Gefahr besteht, dass Fehler ignoriert und Daten beschädigt werden. (**Fehlersignalisierung**)
- möglichst viel Kontextinformation zu einem konkreten Fehler bereitstellen und einfachen Zugriff auf diese Information erlauben, damit die Abstraktionsschicht, an die der Fehler signalisiert wurde, auch dazu in der Lage ist ihn zu interpretieren. (**Fehlerklassifikation**)
- erlauben auf Fehler zu reagieren und eventuell die Fehlerursachen zu beheben. (**Fehlerbehebung**)

3 Fehlersignalisierung

Als Paradebeispiel einer Prozedur welche für bestimmte Eingabe fehlschlägt dient die Funktion atoi[1] aus der Standartbibliothek der Programmiersprache C.

```c
#include <stdlib.h>
int atoi(const char *nptr);
```

Codebeispiel 1: Funktionsprototyp von `atoi`

Diese Prozedur zeichnet sich durch ihre simple Aufgabe, nämlich dem Parsen einer Zeichenkette, welche mit einer Ganzzahl in Dezimaldarstellung beginnen soll, und dem undefinierten Verhalten in Fehlersituationen aus:

> "The functions `atof`, `atoi`, `atol`, and `atoll` need not affect the value of the integer expression `errno` on an error. If the value of the result cannot be represented, the behavior is undefined."[WG199, S. 307]

Es werden Fälle für welche diese Funktion fehlschlagen kann betrachtet und mögliches Verhalten dabei vorgestellt:

- Die Zeichenkette beginnt mit validen ASCII-Zeichen, welche aber nicht numerisch sind.
- Die Zeichenkette beginnt mit einer validen Ganzzahl, welche allerdings nicht mit dem Datentyp `int` ausdrückbar ist.

Die Möglichkeiten die sich dem Implementierer dieser Funktion bieten sind in C stark begrenzt. Er könnte versuchen einen bestimmten Wert aus dem Wertebereich des `int` Datentyps für Fehler zu reservieren, was den Nachteil hätte, dass der Nutzer dieser Funktion ein das erfolgreiche Parsen dieser Zahl nicht mehr von einer Fehlersituation unterscheiden könnte. Eine Alternative wäre es

[1] ASCII to integer

mittels `exit(int)` den Prozess zu beenden, damit dieser Fehler keine negativen Auswirkungen au den weiteren Programmverlauf ergibt. Doch dies würde sämtliche Software welche diese Funktion nutzt alles andere als robust machen. Da der C99-Standard das Verhalten in so einem Fall nicht definiert, wären beide Möglichkeiten bei der Implementation erlaubt. Tatsächlich verhält sich die verbreitete glibc-Bibliothek des GNU-Projekts so, dass beim Lesen nichtnumerischer Zeichen die 0 und beim über-/unterschreiten des Wertebereichs der jeweils größte bzw. kleinste Wert des `int`-Datentyps zurückgegeben wird. Wegen der eben genannten Probleme wird der Einsatz dieser Funktion nicht empfohlen, sondern der einer robusteren Ersatzfunktion:

```
1 #include <stdlib.h>
2 long int strtol(
3         const char * restrict nptr,
4         char ** restrict endptr,
5         int base);
```

Codebeispiel 2: Funktionsprototyp von `strtol` [2]

Im Gegensatz zu `atoi` hat diese Funktion für beide Fehlerfälle ein definiertes Verhalten:

> "If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer."[WG199, S. 311]

Diese Funktion erwartet also eine Referenz auf eine Charpointer-Variable, welche nach dem Aufruf einen Pointer auf das erste nichtnumerische Zeichen enthalten wird. Wenn kein einziges Zeichen geparsed werden konnte, dann gilt also `nptr == *endptr`, woran eine fehlerhafte Eingabe erkannt werden kann. Dass die eingelesene Zahl den Wertebereich überschritten hat, wird allerdings anders mitgeteilt:

> "The [`strtol` function returns] the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, [`LONG_MIN` or `LONG_MAX`] is returned (according to the [...] sign of the value, if any), and the value of the macro `ERANGE` is stored in errno."[WG199, S. 311]

[2] Der Parameter `base` wird ignoriert, und davon ausgegangen dass nur Zahlen in Dezimaldarstellung geparst werden.

Die Wertebereichsüberschreitung wird dem Aufrufer der Funktion also über einen globalen (bzw. in modernen Standartbibliotheken threadlokalen) Seiteneffekt mitgeteilt.

Wie zu sehen ist, stellen sich schon bei einfachen Prozeduren schwere Designentscheidungen zur Fehlersignalisierung.

3.1 Signalisierungspattern

Das `strtol`-Beispiel aus der C-Standardbibliothek hat gezeigt, dass es verschiedene Möglichkeiten gibt um dem Aufrufer einer Prozedur mögliche Fehlerfälle mitzuteilen. Diese verschiedenen Möglichkeiten können als "Pattern" (im Sinne von Lösungsmustern für wiederkehrende Probleme) angesehen werden, welche sich bei der Entwicklung von Bibliotheken in verschiedenen Programmiersprachen herauskristallisiert haben.

3.1.1 Spezielle Rückgabewerte

Dies ist ein verbreitetes Pattern in der C-Programmierung und findet sich in der C-Standardbibliothek häufig wieder. Ein Beispiel wäre die IO-Funktion `fgetc`:

```
1 #include <stdio.h>
2 int fgetc(FILE *stream);
```

Codebeispiel 3: Funktionsprototyp von `fgetc`

Diese Funktion liest das nächste Zeichen/Byte aus einem Stream und liefert es als Rückgabewert. Zu beachten ist, dass der Rückgabewert vom Typ `int` ist, obwohl `unsigned char` für ein Byte ausreichen würde. Dies liegt daran, dass über den Rückgabewert auch mitgeteilt wird ob überhaupt ein Byte gelesen werden konnte. Wenn dies nicht möglich ist (im einfachsten Fall darum, weil der Stream leer ist), dann wird `EOF` zurückgegeben.

Dies wird in C idiomatisch folgendermaßen genutzt:

```c
int byte;
while((byte = fgetc(stream)) != EOF){
  /* process 'byte' */
}
/* stream is empty */
```

Codebeispiel 4: Aufruf von `fgetc`

Die Möglichkeit einen speziellen Wert in Ausnahme- und Fehlerfällen zurückzugeben setzt natürlich voraus, dass der Rückgabetyp der Funktion umfassender ist als die Menge an legitimen Rückgabewerten. Dies ist in statisch typisierten Sprachen nicht immer der Fall, was man an dem Beispiel der Funktion `atoi` erkennen konnte. In dynamisch typisierten Sprachen hingegen, ist es fast immer möglich Werte eines unerwarteten Typs als Rückgabewerte in Fehlersituationen zu verwenden.

Ein Beispiel dafür wäre die Programmiersprache Scheme (R5RS), wo die (sich `atoi` ähnelnde) Prozedur `string->number` existiert. In Fehlersituationen liefert sie einfach `f` (false).

```
(string->number "123")
=> 123
(string->number "-25")
=> -25
(string->number "not a valid number")
=> #f
```

Codebeispiel 5: Interaktion mit `string->number`

3.1.2 Rückgabe einer Nullreferenz

In Programmiersprachen in denen Prozeduren konzeptionell Referenzen (anstatt Werte direkt) als Parameter übergeben bekommen und Rückgabewerte ebenso Referenzen sind (Lisp, Smalltalk, die meisten objektorientierten Skriptsprachen und größtenteils Java/C#), gibt es oftmals eine Nullreferenz welche anstelle jeder anderen Referenz verwendet werden kann. So kann man Nullreferenzen als einen programmiersprachenweiten speziellen Rückgabewert sehen und verwenden.

Diese Nullreferenzen sind häufig als ein “es ist kein Wert vorhanden” zu interpretieren und werden, z.b. in dem Java-Collections-Framework, als Rückgabewert beim Lookup eines nicht vorhandenen Keys-Value-Paares verwendet:

```
Map map = new HashMap(){{
    put("name", "Marie");
    put("age", 17);
}};
if(map.get("surname") == null)
    System.out.println("Map#get(Object key) returned null");
```

Codebeispiel 6: Interface für `java.util.Map`

Problematisch sind Nullreferenzen allerdings (außer dass Nullreferenzen generell verpönt scheinen [Hoa09]) wenn die Collection diese als gültige Werte zulässt. Dies führt zu redundanten Methoden in der Schnittstelle (Java: `Map#containsKey(Object key)`), da man unterscheiden will ob es kein Schlüssel-Wert-Paar gibt oder der Wert einfach die Nullreferenz ist, oder zu Collections welche keine Nullreferenzen aufnehmen (`java.util.Queue`).

3.1.3 Nullobjekt-Pattern [Woo97]

Wenn Voraussicht beim Entwurf von Datentypen gezeigt wird, dann können zu diesen Datentypen spezielle (meist Singleton-) Objekte gehören, welche das Resultat fehlerhafter oder unerlaubter Operationen darstellt. Diese Nullobjekte werden dann anstelle der Nullreferenz als Rückgabewert benutzt werden um einen Fehler darzustellen. Dies wird manchmal in statischen Programmiersprachen verwendet, da ein Nullobjekt das Datentypen sinnvoll erfüllen und/oder Kontextinformation tragen kann. Das Pattern kann praktisch nur bei Wertobjekten welche immutable sind verwendet werden, da nur sehr wenige Programmiersprachen es einem erlauben die Klasse eines Objektes zur Laufzeit zu ändern um darzustellen, dass eine mutierende Operation es invalide gemacht hat.

Eine Beispielanwendung dieses Patterns wäre ein Intervallklasse welche es einem erlaubt zwei Intervalle zu einem neuen zu kombinieren. Dies ist allerdings nur möglich wenn sich diese überlappen.

```
interface Interval {
    public Interval combine(Interval that);
    public boolean isNull();
    ...
}

class IntervalImpl implements Interval {
    ...
    public boolean isNull() { return false; }
    public Interval combine(Interval that) {
        if(that.isnull()) // propagate NotAnInterval instances
            return that;
        if(this.contains(that.begin()) ||
           this.contains(that.end())) // Intervals must overlap
            return new IntervalImpl(min(this.begin(), that.begin()),
                                    max(this.end(), that.end()));
        else
            return new NotAnInterval("Cannot combine " + this +
                                     " and " + that);
    }
}

class NotAnInterval implements Interval {
    ...
    public NotAnInterval(String message) { this.message = message; }
    ...
    public boolean isNull() { return true; }
    public Interval combine(Interval that) { return this; }
}
```

Codebeispiel 7: Das Nullobjekt-Pattern in Java

Hier wird ein Interface für Intervalle definiert mit zwei Implementationen: Eine Konkrete (`IntervalImpl`) und eine für Nullobjekte. Die `combine` Methode kann nun in einem Fehlerfall

eine Instanz von `NotAnInterval` zurückgeben oder, falls sie auf einem `NotAnInterval`-Objekt aufgerufen wurde, das Nullobjekt propagieren.

Zu beachten ist noch die `Interval#isNull()` Methode um valide von nicht-validen Intervallen zu unterscheiden und der Konstruktor für `NotAnInterval`-Instanzen welcher einen `String` mit einer Fehlerursache akzeptiert. In den meisten Fällen wird allerdings nur einzige Instanz eines Nullobjektes erstellt und als Singleton verwendet und der Konstruktor dementsprechend versteckt.

Bei vielen dynamischen Programmiersprachen existiert keine klare Trennung zwischen Nullreferenzen und Nullobjekten, da die programmiersprachenweite Nullreferenz häufig ein Objekt ist welches um die fehlenden Methoden dynamisch erweitert werden kann.

Als Beispiel dient hierfür die Programmiersprache Clojure, wo es möglich ist polymorphe Protokollfunktionen auf der Nullreferenz `nil` zu spezialisieren.

```
1  (defprotocol Vehicle
2    (move [self destination] "move vehicle to destination"))
3
4  (defrecord Car [type driver position]
5    Vehicle
6    (move [this destination]
7      (println driver "drives his" type "from" position "to" destination)
8      (Car. type driver destination)))
9
10 (defrecord Ship [name captain position]
11   Vehicle
12   (move [this destination]
13     (println "The" name "is sailing to" destination ". Aye!")
14     (Ship. name captain destination)))
15
16 (extend-type nil
17   Vehicle
18   (move [_ _]
19     (println nil "is going nowhere!")))
```

Codebeispiel 8: Spezialisierung eines Protokolls auf `nil`

```
(move (Car. :yaris "Aaron" :hamburg) :vegas)
;Aaron drives his :yaris from :hamburg to :vegas
=> #user.Car{:type :yaris, :driver "Aaron", :position :vegas}
(move (Ship. "Black Pearl" "Martin" :port-royal) :tortuga)
;The Black Pearl is sailing to :tortuga . Aye!
=> #user.Ship{:name "Black Pearl", :captain "Martin", :position :tortuga}
(move nil :georgia)
;nil is going nowhere!
=> nil
```

Codebeispiel 9: Aufruf der polymorphen Funktion

Hier sieht man wie eine `move`-Methode auch für die Nullreferenz definiert wurde, und `nil` nun überall dort verwendet werden darf, wo ein `Vehicle` erwartet wird.

3.1.4 Multiple Rückgabewerte

Die bisherigen Fehlerbehandlungsmuster haben eine Eigenschaft gemeinsam: Sie können nicht angewandt werden wenn die Prozedur jeden möglichen Wert zurückgeben könnte, da Rückgabewerte und die Information über den Erfolg des Prozeduraufrufs auf demselben Weg dem Aufrufer mitgeteilt werden.

Dies ist ein wichtiges Designproblem beim Zugriff auf Elemente heterogener Collections, welches von Programmiersprachen und Bibliotheken unterschiedlich gelöst (oder wie im erwähnten Java-Collections-Beispiel, nicht gelöst) wird. Wenn die Programmiersprache erlaubt, dass eine Prozedur nicht nur einen, sondern mehrere Rückgabewerte haben kann, dann kann einfach einer dieser Rückgabewerte über den Erfolg des Aufrufs aussagen.

Eine (relativ) bekannte Programmiersprache die dies zulässt wäre Common Lisp:

```lisp
(defvar *map* (let ((map (make-hash-table)))
                (setf (gethash :name map) "Anna"
                      (gethash :age map) 24
                      (gethash :occupation nil))
                table))
```

Codebeispiel 10: Hashtable in Common Lisp

```lisp
(gethash :name *map*)
=> "Anna"
=> T
(gethash :hobbies *map*)
=> NIL
=> T
(gethash :surname *map*)
=> NIL
=> NIL
> (multiple-value-bind (val exists?) (gethash :surname *map*)
    (if exists?
      (format t "surname is ~a ~%" val) ; print to stdout
      (format t "no surname ~%")))
;no surname
=> NIL
(= (gethash :age *map*) 24)
=> T
```

Codebeispiel 11: Interaktion mit einer Hashtable in Common Lisp

Dies ist ein Beispiel analog zu dem aus den Java-Collections, wo in einer Hashmap die Nullreferenz `nil` als Wert vorkommen kann. Um zu signalisieren dass ein Wert nicht existiert hat die Funktion gethash einen zusätzlichen Rückgabewert, welcher (je nach Vorhandensein) entweder `T` oder `NIL` ist. Um auf diesen zusätzlichen Rückgabewert zuzugreifen existieren special-forms wie `multiple-value-bind` und andere.

Zu beachten ist außerdem, dass Funktionen mit multiplen Rückgabewerten schnittstellenkompatibel sind zu Funktionen mit nur einem Rückgabewert (es wird immer der erste Rückgabewert

der Funktion benutzt). Dies lässt sich nutzen um Funktionen wie `gethash` im Nachhinein mit sinnvollem Verhalten in Fehlerfällen auszustatten ohne dass Konsumentencode angepasst werden muss.

3.1.5 Call-by-reference- und `out`-Parameter

Programmiersprachen, welche zwar keine multiplen Rückgabewerte zulassen, aber über Zeiger oder Referenzen verfügen, bieten ihren Prozeduren die Möglichkeit multiple Rückgabewerte zu simulieren. Dabei deklariert die Prozedur einen Parameter über welchen sie eine Referenz oder einen Zeiger auf eine Variable in der Aufruferprozedur erwartet. Beim Aufruf der Prozedur wird dann in die Variable ein Wert geschrieben.

Eine Prozedur die dies zur Fehlersignalisierung nutzt wäre die Funktion `pthread_create` aus der POSIX-Thread-Bibliothek (in C):

```c
#include <pthread.h>

int pthread_create(
        pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine)(void*),
        void *arg);
```

Codebeispiel 12: Funktionsprototyp von `pthread_create`

```c
pthread_t worker_thread;

switch(pthread_create(&worker_thread, &worker_attr, &worker_func, NULL)) {
case 0: /* success */
        break;
case EAGAIN:
        puts("Cannot create another thread: too many threads already\n");
        break;
case EPERM:
        puts("Cannot create thread: insuffcient permissions\n")
        break;
default:
        puts("Cannot create thread\n");
}

pthread_join(worker_thread, NULL); /* wait for thread to stop */
```

Codebeispiel 13: Ein Aufruf von `pthread_create`

Hier wird also der Rückgabewert von `pthread_create` zur Fehlersignalisierung und der erste Parameter (`pthread_t *thread`) zur Rückgabe einer Datenstruktur an den Aufrufer genutzt.

Eine modernere Programmiersprache welche dieses Pattern immer noch zulässt und in der Standardbibliothek verwendet ist C#:

```c
System.Console.Out.WriteLine("Enter a number.");
string s = System.Console.In.ReadLine();
int value;
if(if (Int32.TryParse(s, out value)) {
        System.Console.Out.WriteLine("You entered " + value);
} else {
        System.Console.Out.WriteLine("That wasn't a number.");
}
```

Codebeispiel 14: `out`-Parameter in C#

Mittels des `out` keywords wird eine (write-only) Referenz auf die Variable `value` übergeben, welche von der `TryParse`-Methode beschrieben wird. Dies wird in den .NET-Bibliotheken von Microsoft häufiger genutzt. Zu den eingebauten Klassen gibt es häufig eine

```
<Klassenname> <Klassenname>.Parse(string s)
```

-Klassenmethode, welche aus einem String den entsprechenden Datentyp erstellt und Fehler über Exceptions signalisiert. Daneben gibt es Klassenmethoden des Schemas

```
bool <Klassenname>.TryParse(string s, out <Klassenname> result)
```

welche Erfolg oder Misserfolg über den Rückgabewert und das Resultat über den `out`-Parameter mitteilt ohne Exceptions zu werfen.

3.1.6 Globale Flags

Eine Alternative für Referenzparameter und multiple Rückgabewerte stellt die Fehlersignalisierung über einen globalen Flag dar. Dabei wird bei einem Fehler in eine festgelegte Variable ein Wert hineingeschrieben, welcher dann vom Aufrufer gelesen werden kann.

Häufig anzutreffen ist dies in der Systemprogrammierung in C, zum Beispiel im Umgang mit Sockets in Unix-Betriebssystemen.

```
int retries = RETRY_LIMIT;
while(true) {
        errno = 0; /* clear error flag */
        if(connect(sock, &remote_server, sizeof(remote_server)) == 0)
                break; /* connection established */

        switch(errno) {
        case ETIMEDOUT:
        case ECONNREFUSED: /* server may just be busy */
                puts("couldn't connect, retrying...");
                sleep(sleep_time);
                sleep_time *= 2;
                retries--;
                if(retries > 0)
                        break; /* retry connecting */
        default: /* other error */
                perror("connect (client)");
                exit(EXIT_FAILURE);
        }
}
```

Codebeispiel 15: Benutzung von `errno`

Hier wird anhand von `errno` unterschieden, ob der Aufruf von `connect` aufgrund eines Programmierfehlers oder fehlenden Berechtigungen fehlgeschlagen ist, oder ob der anzusprechende Server die Verbindung nur nicht angenommen hat. Im letzteren Fall kann es sich lohnen den Verbindungsaufbau nach einer kleinen Wartezeit erneut zu versuchen.

3.1.7 Optiontypes

In vielen statisch typisierten funktionalen Programmiersprachen gibt einen simplen (parametrisierbaren) Datentypen, welcher für das eventuelle Vorhandensein eines Wertes steht.

```
datatype 'a option = NONE
                   | SOME of 'a;
```

Codebeispiel 16: `option`-Typdefinition in Standard ML

Eine Funktion die dies zur Fehlersignalisierung nutzt ist `Int.fromString` aus Standard ML, ist eine weitere Permutation der Möglichkeiten eine Prozedur zu entwerfen, welche einen String zu einer Ganzzahl konvertiert:

```
Int.fromString : string -> int option (* signature of Int.fromString *)

Int.fromString "123";
=> SOME 123 : int option
Int.fromString "Sophie (19)";
=> NONE : int option
```

Codebeispiel 17: Interaktion mit `Int.fromString`

```
fun readNumber () =
  let val _ = print "Enter a number\n"
      val line = valOf (TextIO.inputLine TextIO.stdIn)
      val convertedLine : int option = Int.fromString line
in
  case convertedLine of
    SOME num => print ("You entered " ^ (Int.toString num) ^ "\n")
  | NONE => print "That wasn't a Number\n"
end
```

Codebeispiel 18: Extraktion eines `int` aus `int option`

Beim Aufruf von `Int.fromString line` kann die `Int.fromString`-Funktion also im Fehlerfall den Wert `NONE` zurückgeben oder, bei Erfolg, eine in `SOME` gehüllte Ganzzahl. Der Typ von `convertedLine` ist also `int option`. Dadurch ist den Fall, dass die eingelesene Zeile nicht zu einer Ganzzahl konvertierbar ist, explizit mittels pattern-matching-Syntax zu behandeln.

Der analoge Datentyp dazu in Haskell heißt `Maybe` und ist eine Monade (Instanz der Typklasse `Monad`), wodurch sich fehlerträchtige Berechnungen mittels der `do`-Notation von Haskell verketten lassen:

```
1 getCarLicensePlate name = do
2   person  <- findVal name residentsInCity
3   carList <- getCars person
4   car     <- head carList             -- the head of this list is a car
5   return (getPlate car)
```

Codebeispiel 19: Verwendung von `Maybe` in `do`-Notation

Wir gehen davon aus, dass die Funktionen `findVal`, `getCars` und `getPlate` alle einen in `Maybe` gewrappten Rückgabewert liefern. Die Implementation von `Maybe` als Monade sorgt dafür, dass die Fallunterscheidungen in der 'do'-Notation (genauer gesagt, bei der Verkettung mittels des Operators >>= oder 'bind'-Operators) automatisiert werden.

3.1.8 Fazit

Die bisher vorgestellten Ausnahme- und Fehlersignalisierunspattern zwingen den Aufrufer einer signalisierenden Prozedur dazu, die Ausnahme sofort nach dem Aufruf zu behandeln (mit dem Null-Objekt-Pattern als Ausnahme). Dies beeinträchtigt in den meisten Fällen die Lesbarkeit des Codes, da der reguläre Kontrollfluss mit Abfragen auf Fehler übersät werden muss.

Besonders stark wird damit der funktionale oder expression-orientierte Programmierstil eingeschränkt, da die Rückgabewerte der Funktionen zur Fehlersignalisierung benutzt werden und an Variablen gebunden werden müssen, anstatt direkt als Parameter für andere Funktionsaufrufe zu dienen.

Das größte Problem besteht allerdings darin, dass der Signalisierungscode explizit wiederholt werden muss wenn die Ausnahme in der Aufrufhierarchie weitergeleitet werden soll. Dies erschwert Refaktorisierungen, wie das Auslagern von gemeinsamen Code in eine Unterfunktion, enorm. Desweiteren lassen sich bereits bestehende Prozeduren nicht um die genannten Signalisierungspattern erweitern ohne deren Schnittstellen abzuändern.

Viele dieser Pattern werden in modernen Programmiersprachen deshalb selten benutzt, und von deren Verwendung wird abgeraten, außer in Fällen wo die Ausnahme tatsächlich sofort behandelt werden kann (wie Hashmap-Lookups oder ähnliches) oder das Pattern eine einfache Fehlerpropagierung erlaubt (Null-Objekte oder `Maybe` in Haskell).

3.2 Nicht-lokale Fehlersignalisierung

Das Problem an sämtlichen lokalen Signalisierungspattern besteht darin, dass sie den Aufrufer stets dazu zwingen die Fallunterscheidung zwischen einem erfolgreichem Prozeduraufruf und einem fehlgeschlagenen zu machen. Dies wird durch das Prozedurmodell verursacht, wo der Kontrollfluss an genau einer Stelle in die Prozedur eintreten und an genau einer Stelle sie wieder verlassen kann. Oder anders formuliert: der Kontrollfluss im Erfolgsfall ist derselbe wie im Fehlerfall.

Als Beispiel dient wieder eine String-Ganzzahl-Konvertierungsfunktion (in Scheme):

```
(define (string->integer string)
  (if (valid-integer-string? string)
    (convert-to-int string)
    #f))
```

Codebeispiel 20: Signalisierung über Rückgabewerte

Um diese Funktion robust zu verwenden, muss beim Aufruf stets der Rückgabewert geprüft werden:

```
(let ((converted (string->integer "not an integer")))
  (if converted
    (* converted 10)
    (begin
      (display "can't convert. defaulting to 0")
      0)))
;can't convert. defaulting to 0
=> 0
```

Codebeispiel 21: Fallunterscheidung am Rückgabewert

3.2.1 Errorhandler

Eine Möglichkeit die Fallunterscheidung am Rückgabewert zu vermeiden, wäre es die Funktion um einen Errorhandler-Parameter zu erweitern.

```
(define (string->integer string on-error)
  (if (valid-integer-string? string)
    (convert-to-int string)
    (on-error '(conversion-error string integer))))
```

Codebeispiel 22: `string->integer` mit einem Errorhandler als Parameter

```
(* 10 (string->integer "not an integer"
                       (lambda (error)
                         (display "can't convert. defaulting to 0")
                         0)))
;can't convert. defaulting to 0
=> 0
```

Codebeispiel 23: Aufruf mit einem Errorhandler

Die Technik, eine Prozedur zu übergeben welche im Fehlerfall aufgerufen wird, wird vor allem sehr häufig in Smalltalkdialekten verwendet, da dort anonyme Prozedurliterale ('Blocks') syntaktisch sehr leichtgewichtig sind.

```
| dict |
dict := {#PI -> 3.1415. #E -> 2.7182.} as: Dictionary.

dict at: #Phi ifAbsent: [Transcript show: '#Phi not Found'. nil].
"Transcript: #Phi not found"
nil
```

Codebeispiel 24: Errorhandler in Smalltalkdialekten

Der Codeblock der als Teil der `at:ifAbsent:`-Nachricht gesendet wurde, wird dann von der Methode zur Fehlerbehandlung aufgerufen.

Was diese Technik erlaubt ist die Benennung und Wiederverwendung von Fehlerbehandlungsstrategien, anstatt sie immer wieder anzuwenden. Des weiteren erlaubt dies dem Aufrufer der Funktion von den Fehlersignalisierungsstrategien eine zu wählen die der Situation angebracht ist.

```
(let (on-error (lambda (error)
                 (displayln error)
                 (displayln "Using NaN")
                 +nan.0)) ; use IEEE 754 Not-a-Number
  (+ (string->integer "2" on-error)
     (* (string->integer "5" on-error)
        (string->integer "8; 64" on-error)))))
;'(conversion-error string integer)
;Using NaN
=> +nan.0
```

Codebeispiel 25: Wiederverwendung eines Errorhandlers mittels `let`

Hier wurde eine lokale Funktion definiert, die die Fehlerbehandlung durch die Rückgabe von `NaN` (dem Nullobjekt aller numerischen Datentypen) darstellt.

Diese Errorhandler lassen sich natürlich auch als Parameter weiterreichen und, durch das Aufrufen eines älteren Handlers innerhalb eines neu erstellten, verketten.

```
(define (old-enough? person-string on-error)
  (let ((split (split-at " " person-string)))
    (if (not (= 4 (length split))) ; person-string should consist of
      (on-error '(malformed-input-error)) ; <name> <surname> <job> <age>
      (<= 18 (string->integer (list-ref split 3)
                              (lambda (error)
                                (displayln "can't convert age")
                                (on-error error)))))))) ; next handler
```

Codebeispiel 26: Verkettete Errorhandler

```
(old-enough? "N. Armstrong ex-astronaut 80"
             (lambda (error)
               (displayln "can't verify age")))
=> #t
(old-enough? "N. A. Armstrong ex-astronaut 80"
             (lambda (error)
               (displayln "can't verify age")))
;can't verify age
(old-enough? "N. Armstrong ex-astronaut eighty"
             (lambda (error)
               (displayln "can't verify age")
               +nan.0))

;can't convert age
;can't verify age
=> #f
```

Codebeispiel 27: Interaktion mit `old-enough?`

Continuation-Passing-Style

Das systematische Weiterreichen von prozeduralen Argumenten, welche dann von der Prozedur mit einem aufgerufen werden, wird auch Continuation-Passing-Style (CPS) genannt. Beim vollständigen Transformieren von Programmen in CPS-Form würde keine Funktion jemals einen normalen Rückgabewert haben und normal Zurückkehren, sondern stets eine weitere Funktion (ihre Continuation) aufrufen:

```
(define (cps-string->integer string success-k failure-k)
  (cps-valid-integer-string? string
    (lambda (string-valid?)
      (if string-valid?
        (cps-convert-to-int string success-k failure-k)
        (error-k '(conversion-error string integer))))
    error-k))
```

Codebeispiel 28: Implemantation von `string->integer` im Continuation-Passing-Style

```
(cps-string->integer "20"
                     (lambda (converted) (* 5 converted))
                     (lambda (error) (displayln error) 0))
=> 100
(cps-string->integer "twenty"
                     (lambda (converted) (* 5 converted))
                     (lambda (error) (displayln error) 0))
;'(conversion-error string integer)
=> 0
```

Codebeispiel 29: Aufruf mit expliziten Continuations

Eine volle Konversion zu CPS setzt voraus, dass alle Funktionen einen weiteren prozeduralen Parameter haben, den sie mit ihrem "Rückgabewertäufrufen (die Continuation), und (im Falle der Fehlersignalisierung) dann noch einen welcher bei einem Fehler aufgerufen wird (die Fehlercontinuation). Da dies äußerst umständlich ist, sind reguläre Continuations implizit, d.h. normale Resultate einer Funktion werden als Rückgabewert zurückgegeben.

Für die gleichzeitige Verwendung von impliziten regulären Continuations und expliziten Fehlercontinuations bedarf es allerdings Operationen für nicht-lokalen Kontrollfluss um die implizite Continuation ignorieren zu können.

3.2.2 Explizite Fehlercontinuations

Scheme hat einen Funktion mit dem Namen `call/cc` (`call-with-current-continuation`) und einen analogen Operator `let/cc` welche es ermöglichen eine implizite Continuation in eine explizite zu wandeln (Reifikation). Damit ist es möglich eine Continuation zu erstellen, welche den Abbruch einer Berechnung darstellt.

```
(define the-root-continuation #f)
(let/cc k (set! the-root-continuation k))

(old-enough? "N. Armstrong ex-astronaut eighty"
             (lambda (error)
               (displayln "can't verify age")
               (the-root-continuation (void))))
;can't convert age
;can't verify age
```

Codebeispiel 30: Abbruch von Berechnungen mittels einer Continuation

Dank `the-root-continuation` kann der Kontrollfluss nun in einem Fehlerfalle auch komplett abgebrochen werden, ohne Rücksicht auf einen passenden Rückgabewert.

Ein `let/cc`-Ausdruck kann auch benutzt werden um eine Fehlercontinuation zu erstellen, an die der reguläre Kontrollfluss wieder ansetzt. Dies ist notwendig um Fehler auch tatsächlich zu behandeln zu können.

```
(define (mature-musicians? person-list old-failure-k)
  (let/cc return-k
    (let ((error (let/cc failure-k
                   (return-k
                     ; protected code
                     (all? (lambda (person)
                             (old-enough? person failure-k))
                           person-list)))))

      ;; error hanndling code
      (case (car error)
        ((malformed-input-error)
          (displayln "at least one malformed entry")
          (old-failure-k error)) ; propagate error
        ((conversion-error)
          (displayln "at least one malformed age")
          (return-k 'malformed)) ; return from function
        (else ; propagate error
          (old-failure-k error))))))
```

Codebeispiel 31: Verkettung von Fehlercontinuations

Es wurden mittels `let/cc` zwei Continuations erstellt. Eine, `failure-k` genannt, setzt bei einer Fallunterscheidung für Fehler an und akzeptiert ein Objekt als Fehlerbeschreibung. Die andere Continuation, `return-k`, überspringt die Fallunterscheidung und liefert einen Rückgabewert an den Aufrufer von `mature-musicians?`. Bei der Fallunterscheidung führt ein Konversionsfehler (der in `string->integer` stattfindet) zur Wiederaufnahme des regulären Kontrollflusses, während andere Fehler an die übergebene Fehlercontinuation `old-failure-k` weitergereicht werden.

```
(mature-musicians? '("K. Cobain rockstar 27"
                     "J. Joplin singer 27"
                     "J. Bieber teen idol 17") ; job contains a space
                   the-root-continuation)
;at least one malformed entry
;'(malformed-input-error)
(mature-musicians? '("D. Guetta producer forty-four" ; age isn't numeric
                     "D. Helbig rockstar 20"
                     "P. Pertersen rockstar 20")
                   the-root-continuation)
;Couldn't convert string to integer
;at least one malformed age
=> 'malformed
(mature-musicians? '("V. Jordan rapper 24" ; no error
                     "J. Joplin singer 27"
                     "J. Bieber teen-idol 17")
                   the-root-continuation)
=> #f
```

Codebeispiel 32: Interaktion mit `mature-musicians?`

3.2.3 Implizite Fehlercontinuations

Da explizite Fehlercontinuations sich in der Schnittstelle einer Funktion als ein zusätzlicher Parameter niederschlagen und damit für Schnittstelleninkompatibilität sorgen, ist es von Vorteil ihre Übergabe, genauso wie bei regulären Continuations, implizit zu gestalten. Ein praktisches Mittel um Werte, die über viele Funktionsaufrufe hinweg weitergereicht werden stellen Variablen mit dynamischem Scope dar, welche in Scheme in der Form von Parameterobjekten existieren[Fee03].

```
(define *failure-continuation*
  (make-parameter (lambda (ex)
                    (display "Unhandled error: ")
                    (displayln ex)
                    (the-root-continuation))))
```

Codebeispiel 33: Definition eines Parameterobjektes für Fehlercontinuations

Zur besseren Benutzbarkeit des Parameterobjektes `*failure-continuation*` werden die Prozeduren `throw` und `extend-failure-continuation` definiert, welche darauf zugreifen oder es dynamisch an neue Continuations binden.

```
(define (throw error)
  ((*failure-continuation*) error))

(define (extend-failure-continuation handler thunk)
  (let/cc return-k
    (let ((error (let/cc failure-k
                   (parameterize ((*failure-continuation* failure-k))
                     (return-k (thunk))))))
      (return-k (handler error)))))
```

Codebeispiel 34: Definition von `throw` und `extend-failure-continuation`

Der prozedurale Parameter `thunk` wird also ineinem dynamischen Scope aufgerufen, wo `*failure-continuation*` an eine neue Continuation (`failure-k`) gebunden ist, die ihren Parameter an die Prozedur `handler` übergibt. Dieser Aufruf von `handler` findet allerdings in einem dynamischen Scope, wo `*failure-continuation*` an eine ältere Continuation gebunden ist, statt. Ein Aufruf von `throw` innerhalb von `handler` hätte also keine Endlosrekursion zur Folge. Wenn innerhalb von `thunk` die Fehlercontinuation nicht aufgerufen wird, dann entspricht der Rückgabewert von `extend-failure-continuation` dem von `thunk`.

`extend-failure-continuation` dient also zum Ausführen von Code in einer Ausnahmesituation ohne den regulären Kontrollfluss zu beeinflussen. Dies entspricht der Semantik von vielen einfachen Exceptionsystemen die beliebige Werte als Fehlerbeschreibung akzeptieren. Ein Beispiel für ein solches System wäre das von JavaScript, dessen 'try'-catch'-Syntax mittels eines Makros simuliert werden kann.

```
(define-syntax try
  (syntax-rules (try catch)
    ((try protected ... (catch error handler ...))
     (extend-failure-continuation (lambda (error) handler ...)
                                  (lambda () protected ...)))))
```

Codebeispiel 35: Definition einer `try-catch`-Syntax

```
(try
  (displayln "first 'try' block")
  (try
    (displayln "second 'try' block; before 'throw'")
    (throw 'an-err)
    (displayln "after 'throw'; won't be printed")
    (catch err
      (display "caught ") (display err) (displayln "; (throw 'oth-err')")
      (throw 'oth-err)))
  (catch err
    (display "caught ") (displayln err)
    (if (eq? err 'oth-err)
      'error-caught
      (throw err))))
;first 'try' block
;second 'try' block ; before 'throw'
;caught 'an-err ; throwing 'oth-err
;caught another-error
=> 'error-caught
```

Codebeispiel 36: Nutzung der `try-catch`-Syntax

3.2.4 Fazit

Die Grundlage eines einfachen Exceptionsystems bildet also eine stilisierte Verwendung von dynamisch gebundenen Fehlercontinuations. Dabei wird garantiert, dass die Continuations niemals außerhalb des Scopes in dem sie erstellt wurden, benutzt werden. Somit benötigt man zur Implementation keine vollen, sondern lediglich Escape-Continuations, welche in vielen Programmiersprachen vorhanden sind (`setjmp`/`longjmp` in C als Beispiel).

Der große Vorteil eines Exceptionsystems zur Fehlerbehandlung ist, dass Prozeduren niemals dazu gezwungen werden einen Rückgabewert zu liefern, wenn sie nicht dazu in der Lage sind. Dadurch wird die Programmlogik nicht mit der Fehlerbehandlungslogik vermischt, was den Code lesbarer macht.

3.3 Güte von Signalisierungsmethoden

Besonders viel Wert sollte darauf gelegt werden, dass eine Ausnahmesituation nicht einfach ignoriert wird, da dadurch das Programm in einen Zustand geraten kann, wo Grundannahmen (Invarianten), die die Stabilität vom Rest des Programms garantieren, nicht mehr gelten und dann weitere Fehler auftreten können.

Signalisierungsmethoden, die bei jedem Prozeduraufruf für zusätzlichen Aufwand sorgen, aber gleichzeitig ignorierbar sind, verleiten den Programmierer dazu sie nicht zu beachten. Dies trifft besonders auf "ad-hoc"-Mechanismen wie speziellen Rückgabewerten und Nullreferenzen zu.

Programmiersprachen haben dies, mittels zwei verschieden Strategien, versucht zu verhindern:

1. Sie haben, mittels ihres Typsystems, Ausnahmesituationen nicht-ignorierbar gemacht.

2. Sie haben, durch non-lokalen Kontrollfluss, den Propagationsaufwand komplett beseitigt.

In modernen Programmiersprachen der letzten 20 Jahre wird fast ausnahmslos die zweite Strategie, in der Form von Exceptionsystemen, angewandt. Ausnahmen bilden rein funktionale Programmiersprachen und/oder Programmiersprachen mit nicht-strikten Auswertungsstrategien, da nicht-lokaler Kontrollfluss ein Seiteneffekt ist und nicht mit verzögerter Auswertung zusammenspielt.

Solche Programmiersprachen, wie zum Beispiel Haskell und Clean, haben meist ein ausgeklügeltes statisches Typsystem und verwenden stattdessen Optiontypes 3.1.7. Die Kombination beider Strategien findet sich in der Form von "checked exceptions" wieder. Dabei fließen Exceptions, die beim Aufruf einer Prozedur signalisiert werden könnten, in die Signatur der Prozedur ein. Dies ist im Typsystem der Programmiersprache Java implementiert, und sorgt deshalb für Probleme beim Entwurf polymorpher 2.1 Schnittstellen.

Wenn die Zielsetzung (wie in Kapitel 2 beschrieben) darin besteht, eine Abstraktionsschicht zu signalisieren die möglichst zur Fehlerinterpretation in der Lage ist, dann ist nicht-lokaler Kontrollfluss eindeutig zu bevorzugen, da sonnst dazu verleitet wird Ausnahmen zu früh, und damit nur unzureichend, zu behandeln.

4 Fehlerklassifizierung

Um Fehler- und Ausnahmesituationen zu behandeln muss, nach der Signalisierung des Fehlers, der Fehler in einer passenden Abstraktionsschicht interpretiert werden. Dieser Schritt ist enorm wichtig, da, selbst wenn der Fehler nicht behandelt werden konnte, dadurch eine sinnvolle Fehlermeldung generiert werden kann, oder noch während der Entwicklung des Programms das Debuggen erleichtert wird.

Zur erfolgreichen Interpretation von Fehlern reicht ein gutes Signalisierungsverfahren, welches in der Lage ist an eine möglichst hohe Abstraktionsschicht zu signalisieren, nicht aus[1]. Prozeduren in hohen Abstraktionsschichten verfügen aller Wahrscheinlichkeit nach über einen großen Baum an (direkten oder indirekten) Subprozeduren verfügen und dementsprechend mit vielen verschieden signalisierten Fehlern rechnen.

Deshalb gehen Signalisierungsmethoden stets Hand in Hand mit Klassifizierungsmethoden: Je höher die Abstraktionsschicht liegt, in die ein Fehler signalisiert wurde, desto mehr Information brauchen die Prozeduren dieser Abstraktionsschicht um diesen Fehler von anderen zun unterscheiden. Umgekehrt kann, wenn Information in der Form von "Die Operation ist fehlgeschlagen" vorhanden ist, ein Fehler effektiv nur über einen Prozeduraufruf hinweg signalisiert werden, ohne an Bedeutung zu verlieren.

4.1 Keine Klassifizierung

Die einfachste Form der "Klassifizierung" ist es Fehler nicht zu klassifizieren, sondern nur zu signalisieren. Bei strikt definierten Operationen, wo es nur einen einzigen Fehlerfall gibt, ist dies eine Möglichkeit, sowohl Laufzeit- als auch syntaktischen, Overhead zu sparen. Signalisie-

[1] Die "effektivste" Form einen Fehler zu signalisieren, wäre es `exit(int)` aufzurufen um das aktuelle Programm zu beenden. Als Fehlerbehandlungsstrategie würde dies dem Benutzer allerdings missfallen.

rungsmethoden wie Nullreferenzen 3.1.2, Nullobjekte 3.1.3 und Optiontypes 3.1.7 lassen meistens keine Differenzierung der Fehler zu. Benutzt werden sie daher bei Datenstrukturzugriffen 6 und sonstigen gut definierten Operationen.

Die eines Signalisierung eines unspezifischen Obertypen in Exceptionhierarchien 4.4 bietet ebenso praktisch keine Information über die Fehlerursache kann deshalb ebenso als mangelnde Klassifikation von Fehlern betrachtet werden.

4.2 Errorcodes

Das `errno`-Beispiel 15 aus Abschnitt 3.1.6 zeigt eine Klassifizierung von Fehlern mittels Errorcodes aus der POSIX-Spezifikation[The04, S. 21] die in der Headerdatei `<errno.h>` definiert sind. Errorcodes werden sehr häufig in der Programmiersprache C verwendet, und sind sogar Teil der Standardbibliothek[WG199, S. 186]. Implementiert werden sie als symbolische Konstanten für Ganzzahldatentypen und lassen sich daher nicht von regulären Ganzzahlen unterscheiden. Errorcodes sind nur schwer erweiterbar und müssen von vornherein standardisiert werden. Geschieht dies nicht, dann kann es vorkommen, dass zwei verschiedene Fehler nicht mehr unterscheidbar werden können, weil ihre Konstanten für die selbe Ganzzahl stehen.

Der große Vorteil an Errorcodes liegt in der kompakten Repräsentation eines Fehlers, welcher damit “by-value” über ein CPU-Register Zurückgegeben werden kann. Sie werden deshalb benutzt um Fehler bei Systemcalls zu signalisieren, da es dabei dabei schwierig ist komplexe Datenstrukturen zu übergeben. [2]

Auf den Windows-Plattformen findet sich eine äußerst komplexe Verwendung von Errorcodes, in der Form des Datentypen `HRESULT`, wieder. Dieser Datentyp ist 32 Bit groß, und darf, mittels eines Adressierungsschemas, um weitere Fehlertypen erweitert werden.[Mic11, S. 7]

[2] Da Betriebssystemkernel, wie zum Beispiel Linux auf der x86-Architektur, bei Systemcalls nicht mit programmiersprachenspezifischen Konzepten wie Stacks arbeiten können, müssen alle Parameter und Rückgabewerte über Werte oder Zeiger in (meist begrenzten) CPU-Registern abgehandelt werden. Zeiger müssen dabei auf Speicherbereichsverletzungen überprüft werden, wodurch direkte Werte in Registern wesentlich effizienter sind.

4.3 Symbolische Klassifizierung

Eine Erweiterung von Errorcodes wäre ein Mechanismus, welches garantieren könnte, dass niemals zwei symbolische Konstanten den selben Errorcode hätten. [3] Dies entspricht dem Symboldatentyp, wie er in vielen dynamischen Programmiersprachen vorhanden ist, wo zwei Symbole bei gleichem Namen identitätsgleich sein müssen. In den Scheme-Beispielen aus Abschnitt 3.2 wurden Symbole auf diese Art verwendet, um verschiedene Fehlern voneinander zu unterscheiden. [4]

Ein Beispiel für eine Programmiersprache mit einem Fehlerbehandlungssystem, welches nach diesem Prinzip Fehler klassifiziert, stellt Ada dar. In Ada können symbolische Namen für Fehler mit derselben Syntax wie globale Variablen deklariert werden (ihr "Typ" muss dann `exception` sein):

```
1 -- declare a custom exception
2 Something_Really_Bad_Happened : exception;
```

Codebeispiel 37: Deklaration einer Exception in Ada

Die Fehlersignalisierung ähnelt semantisch der des in Abschnitt 3.2.3 vorgestellten Exceptionssystems, mit dem Unterschied, dass in Ada nicht beliebige Objekte signalisiert werden können, und die Syntax eine Fallunterscheidung mit der Semantik des `case`-Ausdrucks integriert.

[3] Interessanterweise existiert für maschinennahe Programmiersprachen wie C ein solcher Mechanismus: Die Deklaration globaler Konstanten, die in mehreren Quelldateien sichtbar sind (in C: `extern int const symbolic_name;`), garantiert den Konstanten einzigartige Adressen. Somit wird der Linker zur Erhaltung symbolischer Identität missbraucht.

[4] Lispdialekte verfügen über eine äußerst bequeme Syntax für Symbole und einen über Symbole gesteuerten nichtlokalen Kontrollflussmechanismus, welcher die Schlüsselworte `throw` und `catch` popularisiert hat. Allerdings ist der `throw-catch`-Mechanismus explizit *nicht zur Fehlerbehandlung gedacht*, noch ist die Verwendung von Symbolen zur Fehlerklassifizierung idiomatisch.[Moo74, S. 43][SG96, S, 237]

```
procedure Example
is
   Temp : Float; -- to be written by 'Stack_Pop'
begin
   Stack_Pop(Global_Stack, Temp); -- may raise 'Stack_Empty' and 'No_Access'
   if Is_Valid(Temp) then
      raise Something_Really_Bad_Happened;
   end if;
exception
   when err : Stack_Empty | No_Access => -- catches either exception
      Ada.Text_IO.Put_Line("couldn't pop stack");
   when err : Something_Really_Bad_Happened =>
      Ada.Text_IO.Put_Line("something really bad happened");
      raise err;
end Example;
```

Codebeispiel 38: Fallunterscheidung bei Exceptions in Ada

Durch die grenzenlose Erweiterbarkeit um Benutzerdefinierte Fehlertypen, sorgt die Verwendung von Symbolen für detailliertere Fehlerbeschreibungen. Dies kann allerdings auch für Schwierigkeiten bei der Fehlerinterpretation sorgen: Während Code in hohen Abstraktionsschichten bei Errorcodes, aufgrund der Überladung einzelner vordefinierter Codes für unterschiedliche Fehler, Probleme haben könnte Fehler zu unterscheiden, so müsste er bei Symbolen viel zu viele und zu aufwändige Fallunterscheidungen machen, da Symbole nicht in der Lage sind Ähnlichkeiten verschiedener Fehler untereinander auszudrücken.

4.4 Fehlerhierarchien

Es ist häufig sinnvoll Fehler zu kategorisieren. Zum Beispiel sollten `FileNotExistent`-Fehler und `FileNotWritable`-Fehler beim Schreiben einer Datei, zwar voneinander unterscheidbar, aber auch gleichzeitig als Fehler im Zusammenhang mit Dateioperationen zusammenfassbar sein. Die meisten Programmiersprachen mit Fehlerbehandlungssystemen bieten daher auch die Möglichkeit an, Fehler in einer Typhierarchie anzuordnen.

Typischerweise verfügen Programmiersprachen, welche hierarchische Klassifizierung für Fehler anbieten, über ein Klassen- oder ein anderes System für hierarchische Datenstrukturen, welches dann auch für eine Fehlerhierarchie verwendet wird. [5]

Die Klassifikation von Fehlern kann vom Fehlerbehandlungssystem erzwungen werden, indem ein spezieller erweiterbarer Fehlertyp spezifiziert wird und der Signalisierungsmechanismus nur Subtypen dieses Fehlertyps zulässt, oder über nicht erzwungene Konventionen erfolgen. Programmiersprachen die Fehlerklassifikation erzwingen verfügen meistens über spezielle Syntax verfügen um Fehler verschiedener Typen getrennt zu behandeln, während in anderen Programmiersprachen der Programmierer eine Fallunterscheidung manuell durchführen muss.

Als Beispiel für eine Programmiersprache welche Klassifikation erzwingt dient Java:

```
try {
    doOperationThatThrowsVariousExceptions();
} catch(java.io.ConnectException e) {
    // handle connection failure
} catch(java.io.FileNotFoundException e) {
    // handle missing file
} catch(java.io.IOException e) {
    // handle other exceptions that deal with IO
} catch(InterruptedException e) {
    // handle interruption request
} catch(Throwable e) {
    // handle everything else
}
```

Codebeispiel 39: Syntax zur Falluterscheidung von Fehlern in Java

Bei der Signalisierung eines Fehlers wird eine Subtypprüfung zu den angegebenen Fehlertypen in den `catch`-Klauseln durchgeführt. Dies geschieht sequenziell, und wird bei der ersten Klausel, die einen Supertypen des signalisierten Fehlers erwartet, abgebrochen. In der entsprechenden Klausel wird die Variable an das signalisierte Fehlerobjekt gebunden. Wenn keines der Klauseln einen Supertypen dieses Objektes erwartet, dann wird der Fehler automatisch an einen weiter

[5] Dem Autor ist lediglich das Conditionsystem von Common Lisp bekannt, wo die Hierarchie von Fehlertypen unabhängig vom Klassen- und Objektsystem entworfen[Ste90, S. 939]und erst nachträglich integriert wurde.[Pc89]

außen in der Aufrufhierarchie liegendes `try-catch`-Konstrukt resignalisiert. Dabei wird pro `try-catch`-Block maximal eine Klausel ausgeführt und Signalisierungen aus `catch`-Klauseln heraus niemals an Klauseln im selben Block signalisiert. Eine solche Semantik findet sich auch in vielen anderen Programmiersprachen wieder.

JavaScript und andere Skriptsprachen wie Lua und Perl verfügen zwar über einen Signalisierungsmechanismus der dem von Java äquivalent ist, erlauben allerdings die Signalisierung von beliebigen Datentypen. Da diese Sprachen nicht unbedingt über ein standardisiertes Objektmodell verfügen [6] akzeptiert der Signalisierungsmechanismus beliebige Werte als Signal. Deshalb muss die Fallunterscheidung manuell geschehen.

```
try {
    doOperationThatThrowsVariousExceptions();
} catch(e) {
    if(e instanceof iolib.FileNotFoundException) {
        // handle missing file
    } else if(e instanceof iolib.IOException) {
        // handle other exceptions that deal with IO
    } else if(typeof(e) === 'string') {
        alert("error: " + e); // some people throw plain strings
    } else { // all other exceptions
        throw e; // must propagate explicitly
    }
}
```

Codebeispiel 40: Fallunterscheidung von Fehlern in JavaScript

Ein weiterer Vorteil, der bei Nutzung eines Objektsystems zur Fehlerklassifizierung besteht, ist die Möglichkeit Fehlerobjekte mit zusätzlicher Information zu versehen. Ein `IntegerParseError` könnte nicht nur das Scheitern einer String-Ganzzahl-Konversion signalisieren, sondern zusätzlich auch den unparsebaren String als Feld enthalten. Dies würde, selbst wenn der Fehler nicht behandelt wurde, es dem Programmierer erleichtern die Fehlerursache zu finden.

[6] JavaScript verfügt zwar über ein Objektsystem, doch werden manchmal mittels lexical closures alternative Objektsysteme erstellt, welche über eigene Subtypprüffunktionen und Hierarchien verfügen.

4.4.1 Klassifikation mittels Mehrfachvererbung

Obwohl eine Bewertung von Objektsystemen zwar nicht Teil dieser Arbeit ist, findet sich in der Fehlerklassifizierung ein besonders passender Einsatzzweck für Objektsysteme die Mehrfachvererbung unterstützen.[Pit90, Abs. "Classifying Conditions"]

Ein Beispiel für den Vorteil von Mehrfachvererbung wäre ein Fehlertyp für eine Fehlende Datei (`FileNotFound`), einer für fehlende Rechte um sie zu Lesen (`NoPermissionToOpenFile`), einer für eine fehlende Webressource (`HTTPNotFound`) und einer für einen Authentifikationsfehler beim Zugriff auf eine Webressource (`HTTPUnauthorized`).

Bei einem interaktivem Programm wäre es sicherlich von Vorteil, wenn für `NoPermissionToOpenFile` und `HTTPUnauthorized` eine Oberklasse `NoPermission` existieren würde, die in einer einzigen Klausel mit einem Nutzername- und Passwortabfragedialog behandelt werden könnte. Dies wäre eine Klassifikation von Fehlern nach Fehlertyp.

Für einen Webcrawler hingegen wäre es viel praktischer wenn die Fehler nach Subsystem klassifiziert wären, da er dann einfach eine generische `HTTPException` abfangen könnte, und die dazugehörige URL auf eine Blacklist setzen könnte.

Beim Entwurf von "low-level"-Prozeduren und der Fehlerhierarchie für die Fehler die von ihnen signalisiert werden können, ist es, wie in Kapitel 2 erläutert, schwer den genauen Einsatz der Prozeduren vorauszusehen. Und bei einem Klassensystem mit Einfachvererbung erfordert die Wahl des genauen Klassifizierungsschemas Voraussicht auf den spezifischen Verwendungszweck.

Auch die Probleme die Mehrfachvererbung im Zusammenhang mit Implementierungsvererbung verursacht, treten der Modellierung von Fehlerhierarchien viel seltener auf, da Fehlerobjekte meistens nur als Datenstrukturen ohne spezialisierte Operationen dienen.

4.4.2 Reinterpretation von Fehlern

Ein häufige Situation die bei der Fehlerinterpretation auftritt, ist dass der Fehler in einer Abstraktionsschicht zwar Interpretiert werden konnte, aber immer noch keine Fehlerbehandlungsstrategie gewählt werden kann und der Fehler an eine höhere Abstraktionsschicht resignalisiert werden muss. Dabei wäre es von Vorteil, wenn der Fehler mit zusätzlicher Information und einer anderen Klassifikation resignalisiert werden könnte.

Direkt ist dies allerdings nur Programmiersprachen mit sehr dynamischen Objektsystemen möglich, da hierfür der Typ eines Fehlerobjektes dynamisch geändert werden müsste.

```
(defun map-file (mapped-func filename)
  (let ((line-number 0) (result '()))
    (handler-case (do-each-line (line filename)
                    (incf line-number)
                    (push (funcall mapped-func line) result))
      (condition (c) ; catch everything
        (let ((augmented-class ; create augmented type
                (make-instance (class-of (class-of c)) ; get metaclass
                               :direct-superclasses
                               (list (class-of c)
                                     (find-class 'file-error-mixin)))))
          (error (change-class c augmented-class ; resignal as new type
                               :line line-number
                               :filename filename)))))
    (nreverse result)))
```

Codebeispiel 41: Reinterpretation durch Typmutation

Hier wird eine Higher-Order-Funktion `map-file` definiert, die eine übergebene Funktion `mapped-func` mit jeder Zeile einer Datei aufruft und danach eine Liste der Rückgabewerte liefert. Wenn ein Aufruf von `mapped-func` fehlschlägt, dann wäre es für einen Benutzer sehr von Vorteil, wenn er erfahren könnte welche Datei auf welcher Zeile eventuell korrumpiert war. Dafür wird, mittels des Metaobjektprotokolls von Common Lisp [7] , die Klasse des Fehlerobjekts durch eine erweiterte, aber immer noch zur alten kompatiblen, dynamisch erstellten Klasse ersetzt. Der Aufrufer kann dadurch immer noch den alten Fehlertypen (mit dem er *rechnet*) abfangen, verliert aber keinerlei Information bezüglich der genauen Fehlerquelle.

Obwohl durch Typmutation eines Fehlerobjektes die eigentliche Intention der Reinterpretation oder Reklassifizierung eines Fehlers am präzisesten ausdrückbar wäre, kann diese Technik nur in

[7] Obwohl der ANSI Standard von Common Lisp Metaklassen definiert, sind die Folgen eines Aufrufs von `make-instance` auf eine Metaklasse nicht beschrieben. Allerdings ist das Metaobjektprotokoll aus "The Art of the Metaobject Protocol" ein Quasistandard welcher diese Operation auch auf Metaklassen definiert.[GKB91, S. 67]

wenigen Programmiersprachen angewandt werden. In Programmiersprachen mit statischeren Typsystemen werden stattdessen Fehler mit anderen Fehlern parametrisiert.

```
List<Person> parseFile(File f) {
    int linenum = 0;
    try {
        ... // do some iterating and collecting
            result.add(parseString(line));
        ...
    } catch(NumberFormatException e) {
        throw new ParseException(line, f.name(), e); // wrap original
    }
}
```

Codebeispiel 42: Reinterpretation durch Verkettung

Die Signalisierung der `NumberFormatException` wird also abgefangen und stattdessen durch eine `ParseException`, welche (analog zum Beispiel aus Common Lisp) den Dateinamen und Zeilennummer enthält, ersetzt. Dabei wird die die `NumberFormatException` bei der Initialisierung der `ParseException` an den "`Throwable cause`"-Parameter übergeben. Der neue, signalisierte Fehler trägt somit theoretisch sämtliche Information aus der `NumberFormatException` mit sich. [8]

Der große Nachteil dieser Technik ist, dass der Typ der signalisierten Exception nun in keinerlei Relation zum Typ der ursprünglichen Exception steht. Dies hat starke Auswirkungen auf die Behandlung von Fehlern, da nun derselbe Fehlertyp mittels zweier verschiedener Klauseln abgefangen werden muss.

[8] Der andere Grund für die Verwendung von "chained exceptions" liegt an einem Sprachfeature von Java: Bestimmte Fehlertypen ("checked exceptions") wirken sich auf die Signatur von Methoden aus und müssen deshalb, um Schnettstellenkompatibilität zu bewahren, in andere Exceptions verpackt werden. In der Praxis kann es vorkommen, dass die ursprüngliche Exception in über fünf Schaltelungsebenen verpackt ist.

```
void highLevelMethod() {
    ...
    try {
        otherParseCall(); // this signals a plain NumberFormatException
        plist = parseFile(f); // this signals a wrapped NFE
        ...
    } catch(NumberFormatException nfe) {
        // handle NFE
    } catch(ParseException pe) {
        if(pe.getCause() != null) try {
            throw pe.getCause(); // unwrap cause
        } catch(NumberFormatException nfe){
            // handle NFE ...Again!
        } ... // other wrapped errors
    }
}
```

Codebeispiel 43: Fallunterscheidung an verketteten Fehlern

Wie zu sehen ist, sorgt das Verketten von Fehlern für Duplizierung von Fehlerbehandlungscode, was die Wartbarkeit von `highLevelMethod()` beeinträchtigen wird. Die Ursache für dieses Problem liegt wieder, wie bei dem Beispiel mit der Mehrfachvererbung, darin, dass eine "low-level"-Prozedur (`parseFile()`) Vermutungen über ihren Aufrufer gemacht hat. Sie ist in diesem Fall davon ausgegangen, dass ihre Aufrufer mehr an Zeilennummern und Dateinamen als an dem Konversionsfehler interessiert sind. Bewahrheiten sich solche Annahmen nicht, dann wird die Behandlung von Fehlern umständlich.

Die Reinterpretation von Fehlern ist eindeutig nützlich zur Fehlerbehandlung, da im Verlauf des Signalisierungsprozesses sich die Bedeutung eines Fehlers wandeln kann, doch darf ein Mechanismus welcher dies beachtet die ursprüngliche Information nicht Verwerfen. Der `change-class` Mechanismus aus Common Lisp ist zwar in erster Näherung ideal für diesen Einsatzzweck, stellt allerdings eine zutiefst destruktive Modifikationsoperation dar.

4.5 Fazit

Zusammenfassend lässt sich sagen, dass die Klassifizierungsmöglichkeiten die zur Fehlerbehandlung wünschenswert wären, weit über das hinausgehen was in den meisten Objektsystemen angeboten werden kann. Besonders im Bereich der Reklassifikation wären Fortschritte in Objekt- und Typsystemen wünschenswert.

Grundsätzlich stehen zur besseren Klassifikation zwei Ansätze zur Verfügung:

1. Das Objekt- und Typsystem expressiver machen: [9] Der Rest der Programmiersprache würde dann auf die verbesserten Klassifizierungsmöglichkeiten zugreifen können und Fehlerobjekte keinen Sonderstatus in der Programmiersprache erhalten. Es stellt sich hierbei die Frage, ob die Fehlerbehandlung ein ausreichend großer Motivator dafür wäre.

2. Einen unabhängigen Klassifizierungsmechanismus für Fehler entwickeln: Dies sorgt dafür, dass in einer Programmiersprache zwei verschiedene Klassifizierungsmechanismen existieren und Programmierer dazu verleitet werden könnten, den speziellen Fehlerklassifizierungsmechanismus für Zwecke welche nichts mit der Fehlerbehandlung zu tun hätten zu missbrauchen.

[9] Dabei ist das dynamische Typsystem gemeint. Fortschritte im statischen Typsystem helfen nicht, da die Typprüfung, die nach der Signalisierung in höheren Abstraktionsschichten stattfindet, eine dynamische ist. Der Autors rät von Ansätzen mittels statischer Typsysteme generell ab, da dies, wie in Kapitel 2.1 beschrieben, für unnötige Schnittstelleninkompatibilitäten sorgt und die statisch geprüften Typspezifikationen zur Fehlersignalisierung in Java Verbosität hervorrufen.

5 Fehlerbehandlung

Eine Programmiersprache, welche ausgiebige Fehlersignalisierungs- und Klassifizierungsmethoden anbietet, erleichtert die Entwicklung von Software durch klare Fehler, die einem Entwickler beim nachträglichem Debuggen helfen können. In bestimmten Fällen ist es möglich, durch defensive Programmierung und Sicherstellung von Preconditions vor Prozeduraufrufen, sicherzustellen, dass diese Fehler nicht mehr auftreten.

Es gibt allerdings Fehler die aufgrund von äußeren Einflüssen, welche nicht von vornherein abgefragt werden können, entstehen. Ein Beispiel ist das Öffnen von Dateien: Ein Programm kann zwar vor dem Öffnen die Existenz und Leseberechtigung der Datei prüfen, doch ist dadurch der Erfolg einer nachfolgenden Leseoperation nicht sichergestellt, da die Datei von einem anderen Prozess gelöscht, verschoben oder in ihren Berechtigungen modifiziert werden kann.

Diese Fehler können deshalb nicht während der Entwicklung ausgeschlossen werden, und müssen zur Laufzeit behandelt werden.

5.1 Unzulänglichkeiten von `throw-catch`-Exceptionsystemen

Die meisten Programmiersprachen verfügen über eine Variation des in Kapitel 3.2.3 vorgestellten Signalisierungsmechanismus. Dieser Mechanismus schränkt allerdings die Möglichkeiten zur Behandlung von Fehlern ein.

Es dient ein Beispiel mit der `string->integer` Konvertierungsfunktion, welche diesmal mittels `throw` [34] als Signalisierungsverfahren arbeitet.

```
(define (string->integer string)
  (if (valid-integer-string? string)
    (convert-to-int string)
    (throw '(conversion-error string integer))))

(define (parse-integer-file filename)
  (call-with-input-file filename
    (lambda (port)
      (let loop ((line (read-line port)) (result '()))
        (if (eof-object? line)
          (reverse result) ; done
          (loop (read-line port) ; else read next line
                (cons (string->integer line) result)))))))
```

Codebeispiel 44: Parsefunktion mit `throw` als Fehlersignalisierungmethode

Die `parse-integer-file`-Funktion ruft also `string->integer` für jede Zeile der angegeben Datei auf und liefert eine Liste der Konversionsergebnisse als Rückgabewert zurück. Dabei könnte es passieren, dass eine Zeile nicht zu einer Ganzzahl konvertierbar ist. [1]

```
(define (spit filename string) ; writes a datum to a file
  (call-with-output-file filename (curry display string)
                                  #:exists 'replace))

(spit "test" (string-append "123456\n" "2134567\n" "2567\n"))
(parse-integer-file "test")
=> '(123456 2134567 2567)
(spit "test" (string-append "123456\n" "not a number\n" "2567\n"))
(parse-integer-file "test")
;Unhandled error: (conversion-error string integer)
```

Codebeispiel 45: Parsen von Ganzzahlen aus einer Datei

[1] Andere Fehlerfälle, wie das Fehlen einer Datei oder Lesefehler, werden hierbei ignoriert.

Beim Lesen einer Zeile, die nicht zu einer Ganzzahl konvertierbar ist, wird, wie zu erwarten ist, von `string->integer` mittels `throw` ein Fehler signalisiert, welcher zum Abbruch der Berechnung führt.

Doch dies ist nicht unbedingt die Semantik, die ein Aufrufer von `parse-integer-file` haben will. Im Falle einer Mittelwertbildung zur statistischen Auswertung, wäre eine einzige unlesbare Zeile kein vernünftiger Grund um die Auswertung abzubrechen. Dazu kann man den Fehler in der `parse-integer-file`-Funktion abfangen, und einfach die Zeile überspringen:

```
1  (define (parse-integer-file-skip-err filename)
2    (call-with-input-file filename
3      (lambda (port)
4        (let loop ((line (read-line port)) (result '()))
5          (if (eof-object? line)
6            (reverse result) ; done
7            (loop (read-line port)
8                  (try (cons (string->integer line) result)
9                    (catch err ; on conversion-error
10                     (if (equal? (car err) 'conversion-error)
11                       result ; don't cons anything onto result
12                       (throw err)))))))))) ; else propagate
```

Codebeispiel 46: `parse-integer-file` welches Konvertierungsfehler ignoriert

Diese Funktion reagiert auf die Signalisierung von Fehlern aus `string->integer` heraus und kann deshalb fehlerhafte Zeilen überspringen.

Das Problem besteht allerdings darin, dass das Überspringen von Fehlern nicht die einzig mögliche Fehlerbehandlungsstrategie darstellt, sondern unzählig viele Möglichkeiten existieren um den Fehler zu behandeln.

```
(define (parse-integer-file-default-value filename default)
  ...
                  (catch err ; on conversion-error
                    (if (equal? (car err) 'conversion-error)
                      (cons default result) ; use default
                      (throw err)))))))))

(define (parse-integer-file-embed-error filename)
  ...
                  (catch err ; on conversion-error
                    (if (equal? (car err) 'conversion-error)
                      (cons err result) ; embed error in result
                      (throw err)))))))))

(define (parse-integer-file-stop-after-num-errors filename max-err)
  ...
                  (catch err ; on conversion-error
                    (if (< 0 max-error-number)
                      (begin (set! max-err (max-err -1))
                             result)
                      (throw err)))))))))
```

Codebeispiel 47: Verschiedene Fehlerbehandlungsstrategien

Wie zu sehen ist, muss für entweder für jede neue Fehlerbehandlungsstrategie auch eine neue Funktion geschrieben werden, oder der Aufrufer auf die Nutzung einer Funktion verzichten und stattdessen die Datei manuell auslesen. Beide Fälle zeigen wie die prozedurale Abstraktion, unter Berücksichtigung von Fehlerbehandlung, für diese Aufgabe versagt.

5.1.1 Explizite Fehlerbehandlungsstrategien

Der Grund für das Versagen liegt darin, dass die richtige Fehlerbehandlungsstrategie frühestens ab der Abstraktionsschicht des Aufrufers von `parse-integer-file` bekannt ist, aber der Fehler effektiv nur innerhalb der Funktion`parse-integer-file` oder gar `string->integer` behandelt werden kann.

In Scheme ist das Problem durch das Weiterreichen verschiedener Fehlerbehebungsstrategien an höhere Abstraktionsschichten, welche eine bestimmte Strategie wählen können, lösbar.

```
(define (parse-integer-file filename)

  (define (prod-next-result current-line old-result)
    (let/cc return ; call this with the list for the next iteration
      (try (return (cons (string->integer current-line) old-result))
        (catch err
          (throw (list 'parse-integer-error ; create new error
                       (cons 'cause err) ; original error
                       (cons 'line current-line)
                       (cons 'file filename)
                       (cons 'use-val ; recovery strategies
                             (lambda (new-val)
                               (return (cons new-val old-result))))
                       (cons 'skip-error
                             (lambda () (return old-result)))))))))
  ; main looping logic
  (call-with-input-file filename
    (lambda (port)
      (let loop ((line (read-line port)) (result '()))
        (if (eof-object? line)
            (reverse result)
            (loop (read-line port) (prod-next-result line result)))))))
```

Codebeispiel 48: Signalisierung eines Fehlers inklusive Behandlungsstrategien

Der Code ist zwar etwas komplex, stellt aber das beschriebene Konzept dar:

Anstatt sich voreilig auf eine Fehlerbehandlungsstrategie festzulegen, erzeugt die Unterfunktion `prod-next-result` stattdessen ein Fehlerobjekt, welches verschiedene Fehlerbehandlungsstrategien in der Form von anonymen Funktionen anbietet. Da die Funktionen Zugriff auf ihre lexikalische Umgebung haben und Continuations in Scheme eine unbegrenzten Gültigkeitsbereich haben, kann durch den Aufruf der Continuation `return` die Berechnung fortgesetzt werde, obwohl sie durch die Signalisierung eigentlich abgebrochen wurde.

Von der komplexen Implementation von `parse-integer-file` merkt ein Aufrufer nichts, da sie im Erfolgsfalle sogar Schnittstellenkompatibel mit der alten Version 44 ist:

```
(spit "test" (string-append "123456\n" "4321\n" "2567\n"))
(parse-integer-file "test")
=> '(123456 4321 2567)
(spit "test" (string-append "123456\n" "not a number\n" "2567\n"))
(try (parse-integer-file "test")
  (catch err
    (if (eq? 'parse-integer-error (car err))
      ((cdr (assoc 'skip-error (cdr err)))) ; the 'skip-error strategy
      (throw err))))
=> '(123456 2567)
(try (parse-integer-file "test")
  (catch err
    (if (eq? 'parse-integer-error (car err))
      (let ((use-val (cdr (assoc 'use-val (cdr err))))
            (orig-line (cdr (assoc 'line (cdr err)))))
        (use-val orig-line)) ; embed unconverted line into result
      (throw err))))
=> '(123456 "not a number" 2567)
```

Codebeispiel 49: Nutzung von eingebetteten Fehlerbehandlungsstrategien

Die Funktion `parse-integer-file` wurde durch diese Modifikation um ein vielfaches mächtiger und flexibler, ohne dass dabei ihre Schnittstelle erewitert werden musste. Das einzige Problem besteht darin, dass sie in dem Großteil aller Programmiersprachen so nicht implementierbar ist, da die Continuation `return` den dynamischen Gültigkeitsbereich ihres zugehörigen `let/cc`-Ausdrucks verlässt und damit als volle Continuation implementiert werden muss. [2]

[2] Dies führt zu einem Folgeproblem: Die Funktion `call-with-open-file` darf die Datei nicht schließen, bis die Continuation `return` von der Speicherverwaltung freigegeben wurde.[KCR+98, S. 35]

5.1.2 Alternativsemantik für `throw-catch`

Für die Notwendigkeit von Continuations mit unbegrenztem Gültigkeitsbereich ist die Implementation der `try-catch`-Syntax aus Abschnitt 3.2.3 verantwortlich. Dort ist an `*failure-continuation*` stets eine Continuation gebunden, deren Aufruf sofort den dynamischen Gültigkeitsbereich des Signalisiers verlässt. Dies ist nicht unbedingt notwendig, da unmittelbar nach dem nicht-lokalen Sprung nicht beliebiger Programmcode ausgeführt wird, sondern welcher der unter der Kontrolle des Implementierers von `extend-failure-continuation` steht.

```
(define (extend-failure-continuation handler thunk)
  (let/cc return-k
    (let ((error (let/cc failure-k
                   (parameterize ((*failure-continuation* failure-k))
                     (return-k (thunk))))))
      (return-k (handler error)))))
```

Codebeispiel 50: Definition von `extend-failure-continuation` aus Abschnitt 3.2.3

Die Continuation eines Aufrufs von `throw` beinhaltet dabei folgende Funktionsaufrufe:

`throw` ⟶ `failure-k` ⟶ `handler` ⟶ `return-k`

Dabei ist zu beachten, dass `failure-k` eigentlich keinerlei Auswirkungen auf den Wert, der der an `throw` als Parameter übergeben wurde, hat und dadurch ausgelassen werden kann.

`throw` ⟶ `handler` ⟶ `return-k`

Dies gestaltet sich tatsächlich sogar, von Implementierung her, einfacher als die ursprüngliche Definition von `extend-failure-continuation` [3] :

[3] Tatsächlich hat der Autor beim Versuch das, aus anderen Programmiersprachen bekannte, `try-catch`-Konstrukt zu implementieren, den "Fehler" begangen den nicht-lokalen Sprung mittels `failure-k` zu vergessen.

```
(define (extend-failure-continuation handler thunk)
  (let/cc return-k
    (parameterize ((*failure-continuation*
                    (lambda (error)
                      (return-k (handler error)))))
      (return-k (thunk)))))
```

Codebeispiel 51: Redefinition von `extend-failure-continuation` ohne `failure-k`

```
(try
  (displayln "before 'throw'")
  (throw 'error)
  (displayln "after 'throw' ; won't be printed")
  'return-value-of-try
  (catch e
    (displayln "in 'catch'")
    'return-value-of-catch))
;before 'throw'
;in 'catch'
=> 'return-value-of-catch
(try (throw 'err)
  (catch e
    (throw e))) ; resignal e
; heap overflow
```

Codebeispiel 52: `try-catch` mit redefinierter `extend-failure-continuation`

Bei der redefinierten Form von `extend-failure-continuation` ist beim Aufruf einer `catch`-Klausel immer noch die aktuellste Fehlercontinuation an `*failure-continuation*` gebunden, was zur Folge hat, dass Fehler nicht an weiter außen liegende `catch`-Klauseln signalisiert werden, sondern stattdessen an sich selber. Diese Endlosrekursion wurde in der alten Version durch den nicht-lokalen Sprung mittels `failure-k` verhindert, da dadurch `handler` in einem dynamischen Scope, wo `*failure-continuation*` an die alte Fehlercontinuation gebunden war, ausgeführt wurde.

Dieser Effekt lässt sich sehr einfach simulieren, wenn während des Signalisierungsvorgangs `*failure-continuation*` wieder an ihren alten Wert gebunden wird:

```
(define (extend-failure-continuation handler thunk)
  (let/cc return-k
    (let* ((old-fc (*failure-continuation*)) ; capture old value
           (new-fc (lambda (error) ; reinstall old value on signal
                     (parameterize ((*failure-continuation* old-fc))
                       (return-k (handler error))))))
      (parameterize ((*failure-continuation* new-fc))
        (return-k (thunk))))))
```

Codebeispiel 53: Endgültige Definition von `extend-failure-continuation`

```
(spit "test" (string-append "123456\n" "not a number\n" "2567\n"))
(try (parse-integer-file "test")
  (catch err
    (if (eq? 'parse-integer-error (car err))
      (let ((use-val (cdr (assoc 'use-val (cdr err))))
            (cause (cdr (assoc 'cause (cdr err)))))
        (use-val cause)) ; embed error into result
      (throw err))))
=> '(123456 (conversion-error string integer) 2567)
```

Codebeispiel 54: Fehlerbehandlung auf Basis von Escapecontinuations

Diese Definition von `extend-failure-continuation` kann nun im Zusammenspiel mit `parse-integer-error` benutzt werden, ohne dass der dynamische Gültigkeitsbereich der Fehlerbehandlungsstrategien verlassen wird. Alle Continuations werden also als "escape"-Continuations verwendet [4] , wodurch sich das Beispiel auch in Programmiersprachen implementieren ließe, die nur über eingeschränkte nicht-lokale Sprungoperationen verfügen.

Die bis hierhin entwickelten sind demnach strikt mächtiger als die Fehlersignalisierungsmöglichkeiten aus Programmiersprachen mit `throw-try-catch`-Modell, da sie , lassen sich aber

[4] Viele Schemeimplementationen bieten auch Operatoren an, welche eingeschränkte Continuations reifizieren. Mittels Operatoren wie `call/ec` (`call-with-escape-continuation`) und `let/ec` lässt sich daher auf die strikte Verwendung von Continuations innerhalb ihres dynamischen Scopes überprüfen.

dennoch mit den dort vorhandenen Sprachkonstrukten (Funktionsobjekten und nicht-lokalen Sprüngen) implementieren. [5]

Programmiersprachen, die keine Continuations mit unbegrenztem Gültigkeitsbereich reifizieren können, sollten deshalb bei der Signalisierung eines Fehlers keinen voreiligen nicht-lokalen Sprung in eine höhere Abstraktionsschicht durchführen. Dies hängt damit zusammen, dass nur die signalisierende Prozedur sich auf der richtigen Abstraktionsschicht befindet um konkrete Fehlerbehandlungsstrategien anzubieten. Die Abstraktionsschicht an die ein Fehlersignal gesendet wurde, hat dagegen meistens nur zwei Möglichkeiten zur Fehlerbehandlung:

Die aktuellste Operation (in der Hoffnung dass der Fehler nicht noch einmal auftritt) wiederholen oder die aktuelle und sämtliche ausstehenden Operationen abbrechen und den Fehler resignalisieren.

[5] Laut Einschätzung des Autors verlässt in der Implementation auch keine einzige Closure den dynamischen Scope ihrer Erzeugung. Dies bedeutet, dass sich ein solches Fehlerbehandlungssystem auch in Programmiersprachen wie C++, welche nicht über automatische Speicherverwaltung verfügen, implementieren lässt.

6 Analyse der Fehlerbehandlungssysteme von Smalltalk und Common Lisp

Unter den Programmiersprachen, welche über eine relativ große Verbreitung verfügen, haben nur Common Lisp und Smalltalkdialekte ein Fehlerbehandlungssystem, welches die in Abschnitt 2.2 genannten Anforderungen erfüllt. Andere Programmiersprachen haben entweder eine Semantik (`throw-try-catch` 5.1) welche eine präzise Fehlerbehandlung unmöglich macht, oder verfügen nicht über ein standardisiertes Objektsystem zur Klassifizierung von Fehlern.

Beide Programmiersprachen haben gemeinsam, dass sie hochinteraktiv und hochdynamisch sind und dass sie über Programmierumgebungen mit Betriebssystemcharakter verfügen oder verfügten.

6.1 Das Conditionsystem von Common Lisp

Das Conditionsystem von Common Lisp ist Teil des ANSI-Standards und wurde von einem Komitee, welches von Kent M. Pitman geleitet wurde, entworfen und spezifiziert. Dabei ist das Conditionsystem explizit nicht nur ein Fehlerbehandlungssystem, sondern ein System zur allgemeinen Behandlung von Ereignissen die während eines Programmablaufs eintreten können.[Pit01] Desweiteren spezifiziert es die Anwesenheit eines Debuggers, welcher das laufende Programm in bestimmten Situationen anhält.

6.1.1 Fehlersignalisierung

Im Gegensatz zu den bisher behandelten Fehlerbehandlungssystemen hat Common Lisp nicht eine Methode zum Signalisieren von Fehlern, sondern vier verschiedene: `signal`, `error`, `cerror` und `warn`.

Alle diese Funktionen haben mindestens einen Pflichtparameter namens `DATUM`, über welchen sie entweder ein fertiges Conditionobjekt signalisieren oder ein neues Fehlerobjekt erstellen und signalisieren. Alternativ akzeptieren die Funktionen auch einen String, wodurch sie eine neue Condition von einem passenden Standardtypen mit dem String als Nachricht initialisieren.

```
(error (make-condition 'style-warning)) ; use existing object
;Condition STYLE-WARNING was signalled.
;   [Condition of type STYLE-WARNING]

(error 'type-error ; instantiate custom type
       :datum "twenty-five"
       :expected-type 'number)
;The value "twenty-five" is not of type NUMBER.
;   [Condition of type TYPE-ERROR]

(error "something really bad happened") ; use default with message
;something really bad happened
;   [Condition of type SIMPLE-ERROR]
```

Codebeispiel 55: Aufruf von Signalisierungsfunktionen

Das polymorphe Verhalten der Signalisierungsfunktionen scheint hierbei hauptsächlich aus Bequemlichkeitsgründen zu bestehen.

Jede der Funktionen hat eine eigene Signalisierungssemantik, und wird deshalb unterschiedlich benutzt:

- `signal` ist die grundlegendste aller Signalisierungsfunktionen. Wenn mit ihr signalisiert wird, aber keine Funktion in höheren Abstraktionsschichten die signalisierte Condition be-

handelt, dann kehrt `signal` einfach mit `NIL` als Rückgabewert zurück. Erstellt Conditions vom Typ `SIMPLE-CONDITION`, wenn kein Typ angegeben wird.

- `error` ist die Signalisierungsfunktionen welche am ehesten mit dem `throw`-Keyword anderer Programmiersprachen zu vergleichen ist: Wenn mit ihr signalisiert wird und keine Funktion in höheren Abstraktionsschichten die signalisierte Condition behandelt, dann wird laut Spezifikation mittels `invoke-debugger` der interaktive Debugger des Lispsystems aufgerufen. Erstellt Conditions vom Typ `SIMPLE-ERROR`.

- `warn` ist eine Funktion, die im Gegensatz zu allen anderen, nur Conditions, welche ein Subtyp von `WARNING` sind, signalisiert. Wenn eine Warnung signalisiert, aber nicht behandelt wird, dann wird eine textuelle Repräsentation der Warnung an der Fehlerkonsole ausgegeben, und `warn` kehrt mit `NIL` als Rückgabewert zurück. Zusätzlich erstellt die Funktion vor der Signalisierung einen Restart vom Typ `MUFFLE-WARNING`, mit dem die Warnung unterdrückt werden kann. Erstellt standardmäßig Conditionobjekte vom Typ `SIMPLE-WARNING`.

- `cerror` funktioniert grundsätzlich genauso wie `error`, aktzeptiert jedoch ein weiteres Pflichtargument `CONTINUE-STRING`. Vor dem Signalisierungsvorgang wird dann ein Restart vom Typ `CONTINUE` mit der als Parameter angegebenen Beschreibung erstellt. Genauso wie `error` erstellt `cerror` Conditions vom Typ `SIMPLE-ERROR`.

Es ist zu beachten, dass das Signalisierungsverfahren und der Typ einer Condition voneinander entkoppelt sind. Das heißt, dass eine Condition vom Typ `WARNING` mit `error` signalisiert werden kann und, genauso wie jede andere Condition die mit `error` signalisiert wurde, dann den Debugger aufrufen kann.

Zum Abfangen von Conditions stehen in Common Lisp zwei Makros, `handler-case` und `handler-bind`, zur Verfügung.

`handler-case` ist hierbei wenig beachtenswert, da er semantisch genauso funktioniert (und auch ähnlich implementiert 3.2.3 ist[Ste90, S. 972]) wie die aus anderen Programmiersprachen bekannte `try-catch`-Syntax:

```
(handler-case (signal 'file-error)
  (arithmetic-error (e)
    (format t "Error in ~s"
            (arithmetic-error-operands e)))
  (file-error (e)
    "got-file-error"))
=> "got-file-error"
```

Codebeispiel 56: Nutzung von `handler-case`

Genauso wie bei bekannter `try-catch`-Syntax, existieren auch hier Klauseln mit Typangaben, welche der Reihe nach auf Subtyprelationen überprüft werden. Und wie bei herkömmlicher `try-catch`-Syntax wird, vor der Ausführung einer Klausel, ein nicht-lokaler Sprung in den dynamischen Scope des `handler-case`-Ausdrucks durchgeführt.

`handler-bind` hingegen ist ein Makro, welcher deutlich flexibleres reagieren auf Fehler erlaubt:

```
(defvar *dynamic-var* 'NOT-BOUND)
=> *DYNAMIC-VAR*
(handler-bind ((condition
                #'(lambda (c)
                    (format 't "dynamic-var* : ~s~%"
                            *dynamic-var*)
                            nil))) ; don't handle condition
  (let ((*dynamic-var* 'BOUND))
    (warn 'warning) ; signal a condition
    'RETURN-FROM_LET))
;dynamic-var* : BOUND
;WARNING: Condition WARNING was signalled.
=> RETURN-FROM_LET
```

Codebeispiel 57: Nutzung von `handler-bind`

Mittels `handler-bind` kann also auf signalisierte Conditionobjekte reagiert werden, ohne sie behandeln zu müssen. Dazu muss die Handlerfunktion einfach regulär zurückkehren. Dabei

findest die Ausführung der Handlerfunktion im dynamischen Scope des Signalisierers statt, was man am Wert der Variable `*dynamic-var*` erkennen konnte. Des weiteren, da die Handler direkt als Funktionen übergeben werden, ist es möglich Handlerfunktionen an einer Stelle zu definieren und mehrmals wiederzuverwenden.

Dabei ist zu beachten, dass Handlerfunktionen zwar im dynamischen Scope des Signalisierers aufgerufen werden, Resignalisierung von den Handlerfunktionen aus allerdings nur an Handler möglich ist, die außerhalb des dynamischen Scope des entsprechenden `handler-bind`-Ausdrucks liegen. Diese Semantik entspricht der in Abschnitt 5.1.2 vorgestellten und wird auch "Condition Firewall"[Pit01] genannt. Sie verhindert Endlosrekursion beim resignalisieren von Conditions:

```
(handler-bind ((error #'(lambda (c) ; catch errors on the outside
                          (error 'warning)))) ; resignal as warnings

  (handler-case (error 'error) ; signal error on the inside
    (warning (c) ; catch warnings on the inside
      (format t "Everything is fine!~%"))))
;Condition WARNING was signalled.
;   [Condition of type WARNING]
```

Codebeispiel 58: Die "Condition Firewall" von Common Lisp

Der Signalisierungsmechanismus von Common Lisp erlaubt damit nur die Signalisierung von niedrigen Abstraktionsschichten in höhere, aber nicht umgekehrt.

6.1.2 Fehlerklassifizierung

Das Conditionsystem von Common Lisp erlaubt nur das Signalisieren von Objekten aus einer besonderen Fehlertyphierarchie, wo alle Fehlertypen Subtypen der Klasse `CONDITION` sind.

Dabei gibt es zwei besondere Subklassen von `CONDITION` namens `WARNING` und `SERIOUS-CONDITION`. `WARNING` stellt den Supertypen aller Conditions dar, welche mittels `warn` als Warnung signalisiert werden können. Dabei wird dies von der Funktion `warn`, wie inm vorherigen Abschnitt 6.1.1 beschrieben, auch erzwungen. `SERIOUS-CONDITION` ist der Supertyp aller Conditions deren Signalisierung ein Ereignis darstellt, welches schwerwiegend genug ist dass es behandelt wenden

muss. Alle Conditionobjekte, welche Instanzen von `SERIOUS-CONDITION` sind sollten mittels `error` oder `cerror` signalisiert werden. Dies wird allerdings nicht erzwungen.[Ste90, S. 994]

Auffällig an der Conditionhierarchie ist, dass die Fehlerobjekte über enorm wenig Funktionalität verfügen. So verfügen Instanzen der Wurzelklasse der Fehlerhierarchie weder über irgendwelche Instanzvariablen/Slots noch gibt es Methoden welche auf `CONDITION` spezialisiert sind (außer `print-object`). Das Hauptunterscheidungsmal zweier Conditionobjekte ist somit ihr Typ.

Obwohl das Conditionsystem mit dem Common Lisp Object System (*CLOS*) integriert ist, und Conditionobjekte und -klassen reguläre CLOS-Objekte und -Klassen sind, existiert zur Definition von Conditionklassen ein eigener `define-condition`-Makro:

```
1 (define-condition parse-integer-error (error)
2   ((line :initarg :line) (file :initarg :file))
3   (:report (lambda (c s)
4              (format s "Can't parse line \"~A\" in file '~A'"
5                      (slot-value c 'line) (slot-value c 'file)))))
```

Codebeispiel 59: Definition einer Conditionklasse

Die Fehlerhierarchie lässt Mehrfachvererbung zu, wodurch wichtige Aspekte voneinander getrennt werden können. So verfügen nicht alle Conditiontypen über eine Fehlernachricht, sondern nur Subtypen von `SIMPLE-CONDITION`. Da die meisten Implementierungen das in "The Art of the Metaobject Protocol"[GKB91] beschriebene Metaobjektprotokoll implementieren, lassen sich auch komplett neuartige Klassifizierungsmöglichkeiten implementieren, wenn man eine neue Klassenhierarchie mit einer eigenen Metaklasse erstellt.

Common Lisp bietet also sehr mächtige Klassifizierungsmöglichkeiten für Fehler und Ereignisse an, indem es dafür sein äußerst mächtiges Objektsystem verwendet.

6.1.3 Fehlerbehandlung

Common Lisp bietet einen Fehlerbehandlungsmechanismus in der Form von "Restarts" an. Dieser Mechanismus ist von dem Signalisierungsmechanismus getrennt, und kann unabhängig verwendet werden.

Restarts sind, vom Konzept her, Variablen mit dynamischen Scope die an benannte Escapecontinuations gebunden werden. Dabei stellen sie zusätzlich auch eine Schnittstelle zum Debugger dar.

Die einfachste Möglichkeit einen Restart anzulegen stellt der Makro `with-simple-restart` dar. Dabei erwartet der Makro einen Namen (als Symbol) und eine Beschreibung (als Formatierungsstring).

```
(with-simple-restart (outer-restart "I'm outside")
  (with-simple-restart (inner-restart "I'm inside")
    (invoke-debugger (make-condition 'condition))
    (format t "After 'invoke-debugger'~%"))
  (format t "After inner restart~%"))
; Debugger:
;Condition CONDITION was signalled.
;   [Condition of type CONDITION]
;
;Restarts:
; 0: [INNER-RESTART] I'm inside
; 1: [OUTER-RESTART] I'm outside
; 2: [RETRY] Retry SLIME REPL evaluation request.
; 3: [*ABORT] Return to SLIME's top level.
;
; Clicking on Restart no. 1:
=> NIL
=> T
```

Codebeispiel 60: Interaktion mit Restarts im Debugger

Restarts werden also als Debugaktionen mit Namen und Beschreibung dargestellt, welche beim Ausführen den Debugger beenden und den Kontrollfluss bei dem entsprechenden `with-simple-restart`-Ausdruck (oder anderen Ausdrücken, welche Restarts erstellen) fortsetzen. Dabei hat dieser Ausdruck potentiell zwei Rückgabewerte: Einer welcher den normalen Rückgabewert des Ausdrucks darstellt und ein zusätzlicher welcher `T` ist wenn Restart benutzt wurde.

Restarts lassen sich aber auch vom Programmatisch ausführen. Dafür existiert die Funktion `invoke-restart`:

```
(with-simple-restart (outer-restart "I'm outside")
  (with-simple-restart (inner-restart "I'm inside")
    (invoke-restart 'inner-restart)
    (format t "After 'invoke-debugger'~%"))
  (format t "After inner restart~%"))
;After inner restart
=> NIL
```

Codebeispiel 61: Aufruf von `invoke-restart`

Einzelne Restarts lassen sich also über ihren Namen voneinander unterscheiden, und dann Aufrufen, wobei die Namen, anders als Conditionstypen, nicht in einer Hierarchie angeordnet sind.

Dies wird im Zusammenspiel mit `handler-bind` zur Fehlerbehandlung genutzt. Dabei kann eine Handlerfunktion wegen der "Condition Firewall" zwar keine Conditionobjekte an eine "low-level"-Prozedur signalisieren, kann aber, da Restarts nicht von der "Condition Firewall" betroffen sind, Fehlerbehandlungsstrategien die als Restarts implementiert wurden, aufrufen.

Da der einfache nicht-lokale Kontrollflusstransfer, der von `with-simple-restart` angeboten wird, nur eine von vielen möglichen Fehlerbehandlungsstrategien darstellt, bietet Common Lisp mit `restart-case` und `restart-bind` noch zwei weitere Makros an, mit denen Restarts erstellt werden können.

`restart-case` ähnelt syntaktisch stark dem `handler-case`-Makro 56 und erlaubt es mehrere verschiedene Restarts gleichzeitig anzulegen. Damit lässt sich ausgezeichnet die Funktion `parse-integer-file` 48 aus Abschnitt 5.1.1 implementieren.

```
(defun read-new-value () ; create a value fom user input
  (format t "Enter a value: ")
  (multiple-value-list (eval (read))))

(defun parse-integer-file (filename)
  (let ((result '()))
    (with-open-file (file filename)
      (do* ((line (read-line file nil) (read-line file nil)))
           ((eq nil line))
        (let ((converted (parse-integer line :junk-allowed t)))
          (if converted
              (push converted result)
              (restart-case (error 'parse-integer-error
                                   :file filename
                                   :line line)
                (use-value (value)
                  :interactive read-new-value
                  :report "Use a replacement value."
                  (push value result))
                (continue ()            ; do nothing
                  :report "Skip malformed line."))))))
      (nreverse result))))
```

Codebeispiel 62: Definition von `parse-integer-file` in Common Lisp

Vor der Signalisierung eines Fehlers wurden also mittels `handler-case` zwei Restarts als Fehlerbehandlungsstrategien angelegt, welche denen aus dem Originalbeispiel 48 in Scheme entsprechen. Ein konzeptioneller Unterschied besteht darin, dass in Common Lisp die Fehlerbehandlungsstrategien nicht Teil des Fehlerobjektes sind, sondern unabhängig in der Form von Restarts existieren. Desweiteren ist dieses Fehlerbehandlungssystem auch auf die interaktive Nutzung ausgelegt:

```
(spit-strings "test" "123456" "twenty" "2567")
=> NIL
(handler-bind ((parse-integer-error
                 #'(lambda (c) (invoke-restart 'continue))))
  (parse-integer-file "test"))
=> (123456 2567)
(parse-integer-file "test")
; Debugger:
;Can't parse line "twenty" in file 'test'
;   [Condition of type PARSE-INTEGER-ERROR]
;
;Restarts:
; 0: [USE-VALUE] Use a replacement value.
; 1: [CONTINUE] Skip malformed line.
; 2: [RETRY] Retry SLIME REPL evaluation request.
; 3: [*ABORT] Return to SLIME's top level.
;
; Clicking on Restart no. 0:
;Enter a new value: 20
=> (123456 20 2567)
```

Codebeispiel 63: Interaktion mit `parse-integer-file`

In Common Lisp stellt der Programmierer/Benutzer also die Abstraktionsschicht dar, welche, im Fall dass alle anderen den Fehler nicht interpretieren wollen/können, zur Interpretation und Auswahl einer Fehlerbehandlungsstrategie aufgefordert wird. Dieses Modell erlaubt es, wenn der Debugger durch eine Benutzerfreundliche GUI ersetzt wird [1] , das Conditionsystem als interaktive Fehlerbehandlung in einer Software für Endbenutzer zu nutzen.

Zusätzlich zum Aufruf eines Restarts gibt es auch Funktionen wie `compute-restarts` und `find-restart`, welche zum reflektiven Zugriff auf Restarts dienen und sie als "first-class"-Objekte reifizieren, dies wird vom Debugger genutzt um sie darzustellen, kann aber auch von regulärem Programmcode genutzt werden. Eine Handlerfunktion kann zum Beispiel überprüfen

[1] Common Lisp hat hierfür eine Variable (`*debugger-hook*`) welche an eine eigene Debuggerimplementierung gebunden werden kann.

ob ein Restart mit dem gewünschten Namen überhaupt existiert und, falls nicht, eine Alternative Fehlerbehandlungsstrategie wählen.

6.1.4 Fazit

Das Conditionsystem von Common Lisp setzt die Anforderungen aus Abschnitt 2.2 durch drei, voneinander relativ unabhängige, Mechanismen um. Die vier Signalisierungsfunktionen funktionieren unabhängig vom konkreten Conditiontyp und können, durch die Übergabe eines Strings, ohne vorherige Erstellung von Conditions und Conditionhierarchien verwendet werden. Restarts können angelegt und nie aus Handlerfunktionen aufgerufen werden, wodurch sie immer noch nützlich zum Debuggen sind

Obwohl die drei Mechanismen angepasst sind um die drei Anforderungen perfekt zu erfüllen, existieren bei den Mechanismen starke Überschneidungen: Restarts und Handler stellen grundsätzlich sehr ähnliche Mechanismen dar (Suche im dynamischen Scope und Sprünge), Sind allerdings getrennt implementiert, wobei beide Einschränkungen haben (Handler können nur in eine Richtung signalisieren, Restarts sind nicht in einer Hierarchie angeordnet). Auch das Anbieten von vier verschieden Funktionen zur Signalisierung, zwei verschiedenen Makros zum Abfangen, einer Besonderen Syntax zur Definition und Instanziierung von Conditions sowie drei Makros zum Anlegen von Restarts, lässt die Frage aufkommen, ob die Mechanismen nicht vereinheitlicht werden könnten.

6.2 Das Exceptionsystem in Smalltalk

Dass das Fehlerbehandlungssystem von Smalltalk als “Exceptionsystem” bezeichnet wird, lässt auf den ersten Blick vermuten dass es ähnliche Einschränkungen wie die Exceptionsysteme von Programmierprachen wie Java oder Python haben müsste. Es ist allerdings, genauso wie das Conditionsystem von Common Lisp, ein Fehlerbehandlungssystem welches auch in der Lage ist Fehler zu behandeln, anstatt Berechnungen als ersten Schritt abzubrechen.

6.2.1 Fehlersignalisierung

Das Signalisieren von Exceptionobjekten in Smalltalk erfolgt grundsätzlich durch das Instanziieren des Objektes mittels Nachrichten an die gewünschte Exceptionklasse und dem anschleßenden Senden der `signal`-Machricht an das Exceptionobjekt. Alternatv dazu ist es auch möglich `signal` an das Klassenobjekt zu senden, was eine Conveniencemethode für das Erstellen einer neuen Exception und dem sofortigen Senden ist. Auch für das Abfangen einer signalisierten Exception existiert keine spezielle Syntax, sondern nur `on:do:` Nachrichten die an `BlockClosure` Instanzen gesendet werden:

```
1 [(1 + 2 + 3 + 4) / 0]
2     on: ZeroDivide
3     do: [:e|
4         {e.'tried to divide '.e dividend.' by zero'} concPrintLn.
5         e return: #divByZero.
6 ].
7 "TS: ZeroDivide: tried to divide 10 by zero"
8 => #divByZero
```

Codebeispiel 64: Exceptionhandler in Smalltalk

Das Senden einer `on:do:`-Nachricht an einen Block sorgt also für eine sofortige Auswertung des Blocks [2] , wobei alle Exceptions die während dieser Auswertung signalisiert werden und Instanzen der mitgesendeten Klasse sind, an den Handlerblock übergeben werden.

Was Exceptionobjekte in Smalltalk von Exceptions aus anderen Exceptionsystemen unterscheidet sind die Nachrichten die an sie gesendet werden können. In dem letzten Beispiel wurde die `return:`-Nachricht gesendet, welche das Exceptionobjekt dazu veranlasst das Verhalten typischer `try-catch`-Blöcke zu simulieren und den gesendeten Parameter deshalb als Rückgabewert des (mittels `on:do:`) geschützten Blocks zu verwenden. Dabei ist zu beachten, dass das Exceptionobjekt für den Signalisierungsprozess selber verantwortlich ist, weiß welcher Handlerblock gerade ausgeführt wird, und deshalb auf die `return:`-Nachricht korrekt reagieren kann.

[2] Die Methode für `on:do:` muss tatsächlich nur `value` an den Block senden, wobei bei Smalltalkimplementationen mit Tail-Call-Optimization(TCO) das Verwerfen des Aktivierungskontextes unterdrückt werden muss.

Der Signalisierungsvorgang ist vollständig über den reflektiven Zugriff auf die Aktivierungsframes mittels der Pseudovariable `thisContext` implementierbar und auch so implementiert.

```
| ondo raiseExn |
ondo := BlockClosure lookupSelector: #on:do:. "the on:do: method"

raiseExn := [:exn | | ctx |
       ctx := thisContext. "reify current activation frame"
       [ctx := ctx sender] "find handler block"
                doWhileFalse:
                [ctx method == ondo and:
                    [exn isKindOf: ctx arguments first]].

        exn privHandlerContext: ctx.
        ctx arguments second value: exn. "invoke handler block"
].

[raiseExn value: ZeroDivide new] on: ZeroDivide do: [:e |
        'a Hello from the handler block' printLn.
        e return: 'used Exception to return from Handler'.
].
"TS: a Hello from the handler block"
=> 'used Exception to return from Handler'
```

Codebeispiel 65: Fehlersignalisierung per Reflection

Hier wurde (aus Gründen der besseren Darstellbarkeit) ein Block namens `raiseExn` erstellt, welcher einen Großteil des Signalisierungsvorgangs implementiert. Dabei werden die Aktivierungsframes der aktuellen Berechnung durchlaufen, bis sich ein Aktivierungskontext findet, der durch eine `on:do:`-Nachricht ausgelöst wurde und wo das erste Argument (der Exceptiontyp) einen Supertyp der übergebenen Exception darstellt. Der Exception kann dann mit diesem Kontext instanziiert und der Handlerblock (das zweite Argument der `on:do:`-Nachricht) mit ihr aufgerufen werden. Da der Signalisierungsvorgang so einfach implementierbar ist, lässt er sich in eigenen Exceptionklassen nach Wunsch anpassen.

Die eingebauten Exceptions in Smalltalk verstehen neben `signal` und `return:` auch weitere Nachrichten zur Steuerung des Signalisierungsvorgangs. Eine interessante, vor allem als Kontrast zum Conditionsystem von Common Lisp, ist die Resignalisierungsnachricht `resignalAs:`.

```
Exception subclass: #SignalA.
Exception subclass: #SignalB.
[
        [       'outer block (1)' printLn. "1"
                [        'middle block (2)' printLn. "2"
                        [        'inner block (3)' printLn. "3"
                                SignalA new signal.
                        ] on: SignalB do: [:e |
                                'inner handler (5)' printLn. "5"
                                e resignalAs: SignalA new.
                        ]
                ] on: SignalA do: [:e |
                        'middle handler (4)' printLn. "4"
                        e resignalAs: SignalB new.
                ]
        ] on: SignalB do: [:e |
                'outer handler' printLn. "never called"
        ]
] on: Exception do: [:e |
        {'Exception '.e.' was not handled (6)'} concPrintLn. "6"
].
"TS: outer block (1)"
"TS: middle block (2)"
"TS: inner block (3)"
"TS: middle handler (4)"
"TS: inner handler (5)"
"TS: Exception SignalA:  was not handled (6)"
```

Codebeispiel 66: Resignalisierung in Smalltalk

Es fallen hier zwei Sachen auf: Zum Ersten gibt es in Smalltalk nicht die aus Common Lisp bekannte “condition firewall”, da ein `SignalB` aus dem mittleren Handler an den inneren und

nicht den äußeren Handler signalisiert wurde. Die zweite Auffälligkeit ist, dass danach ein `SignalA` aus dem inneren Handler signalisiert wird, welcher aber nicht vom mitleren Handler abgefangen wird.

In Smalltalk werden rekursive Handleraufrufe also verhindert ohne die Signalisierungsrichtung zu beschränken. Dies geschieht feingranular durch das setzen einer Instanzvariable im Aktivierungsframe der `on:do:`-Nachrichten, welche beim Signalisierungsvorgang abgefragt wird. Jeder Handler kann damit nur ein einziges mal pro Signalisierungsprozess aufgerufen werden und eine Rekursion findet nicht statt.

Eine weitere Resignalisierungsmethode findet sich in den `pass`- und `outer`-Nachrichten wieder: Diese sorgen für eine Resignalisierung an weiter außen liegende Handler und verfügen, die Fehlerbehandlung ausgenommen, über die selbe Semantik wie das Resignalisieren von Exceptions in Sprachen wie Java. Allerdings ist es mittels der `isNested`-Nachricht möglich, bevor resignalisiert wird auf die Existenz umliegender Handler, welche den Fehler abfangen würden, zu prüfen.

6.2.2 Fehlerbehandlung

Das Protokoll von Exceptionobjekten enthält drei Nachrichten zur Fehlerbehebung: `retry`, `retryUsing:` und `resume:`.

```
| dir file |

file := [dir entryAt: 'test'] on: MessageNotUnderstood do: [:e |
        dir := FileList modalFolderSelector.
        e retry.
].

file := [dir entryAt: 'test'] on: MessageNotUnderstood do: [:e |
        e retryUsing: [FileSystem find: 'test'].
].
```

Codebeispiel 67: `retry` und `retryUsing:`

Die `retry`- und `retryUsing:`-Nachrichten führen den geschützten Block nochmal, oder einen Alternativblock anstelle dessen aus. Diese Operationen sind zwar praktisch, lassen sich aber auch in gewöhnlichen Exceptionsystemen mittels Flags und Schleifen realisieren. [3]

`resume:` führt hingegen einen Sprung zum ursprünglichen Signalisierungskontext aus und gibt das übergebene Argument an den Sender der ursprünglichen `signal`-Nachricht zurück:

```
1 [{3. 6. 9. 0. 25} collect: [:num | 1 / num]]
2         on: ZeroDivide
3         do: [:e | e resume: nil].
4 => {(1/3). (1/6). (1/9). nil. (1/25)}
```

Codebeispiel 68: Wiederaufnahme einer Berechnung

Hier wurde also ein Handler angelegt, welcher Fälle einer Division durch Null abfängt und durch das Nullobjekt `nil` ersetzt. Mit dieser Nachricht sind mächtige Fehlerabstraktionmöglichkeiten umsetzbar. Zum Beispiel kann man Berechnungskontexte erstellen, wo unbekannte Nachrichten still ignoriert werden:

[3] Wenn ein `goto`-Statement in der Sprache fehlt.

```
BlockClosure>>swallowMNU
        ||
        ^self on: MessageNotUnderstood do: [:e | e resume: nil]

[anObject aMessage anotherOne and more.] swallowMNU.
=> nil

BlockClosure>>badIdea
        | string |
        string := ''.
        self on: MessageNotUnderstood do: [:e |
                string := string, ' ', (e message selector asString).
                e resume: nil].
        ^string

[1 I heard that you like Domain Specific Languages.]
        badIdea.
=> ' I heard that you like Domain Specific Languages'
```

Codebeispiel 69: `resume:` von `MessageNotUnderstood`

Normalerweise kehrt durch eine `resume:`-Nachricht die Ausführung wieder an den ursprünglichen Signalisierungskontext zurück, aber durch die Nachricht `outer` lässt sich die Exception so resignalisieren, dass eine `resume:`-Nachricht den Kontrollfluss wieder in dem Handlerkontext wo `outer` gesendet wurde ansetzt. Um zum ursprünglichen Signalisierungskontext (oder dem letzten Kontext wo `outer` gesendet wurde) zurückzukehren, muss noch einmal `resume` gesendet werden.

```
Warning subclass: #ASignal.
[       [       'before signal (1)' printLn. "1"
                ASignal new signal.
                'after second resume (5)' printLn. "5"
        ] on: ASignal do: [:e |
                'before outer (2)' printLn. "2"
                e outer.
                'after first resume (4)' printLn. "4"
                e resume.
        ].
] on: ASignal do: [:e |
        'before first resume (3)' printLn. "3"
        e resume.
]
"TS: before signal (1)"
"TS: vbefore outer (2)"
"TS: before first resume (3)"
"TS: after first resume (4)"
"TS: after second resume (5)"
=> nil
```

Codebeispiel 70: Semantik von `outer`

6.2.3 Fehlerklassifizierung

Exceptionklassen in Smalltalk sind Teil der regulären Klassenhierarchie und definieren ihr Verhalten über vier Hookmethoden:

- Die Methode für `isResumable` gibt an, ob es erlaubt ist die Berechnung im Signalisierungskontext wieder fortzusetzen. Sie wird beim Empfangen von `resume`- und `resume:`-Nachrichten abgefragt, wird aber ignoriert wenn `resumeUnchecked:` gesendet wird. Exceptions welche `isResumable` mit `false` beantworten stellen Berechnungen dar, welche auf einen schwerwiegenden Fehler gestoßen sind. Die Defaultimplementierung ist `^true`.

- Die Methode für `defaultAction` wird aufgerufen, wenn der Signalisierungsprozess keine passenden Handler gefunden hat, wodurch sie die Seriosität der jeweiligen Exception angibt. Eine Implementierung, welche einfach `nil` zurückgibt sorgt für das Fortsetzen der Berechnung nach der Signalisierung und ist dementsprechend nur ein Signal und kein Fehler. Wohingegen Warnungen durch die Ausgabe einer Mitteilung und Fehler durch das Aufrufen des Debuggers implementiert werden. Die Defaultimplementierung ist der Aufruf des Debuggers.

Daneben gibt es noch Methoden für `defaultResumeValue` und `defaultReturnValue`, welche allerdings uninteressant sind, da sie nur angeben was beim Senden der argumentlosen Varianten der `return:-` und `resume:`-Nachrichten geschieht.

Die Exceptionhierarchie hat zwei größere Subhierarchien, die zu beachten sind: `Error` stellt die Oberklasse von Fehlern dar und implementiert die Methode zu `isResumable` mit `^false`. Dabei ist allerdings zu beachten, dass viele Subklassen wie `ZeroDivide` und `MessageNotUnderstood` eine `resume:`-Nachricht wieder zulassen. Die andere Subklasse heißt `Notification` und implementiert die Hookmethode zu `defaultAction` mit `^nil`. Das bedeutet, dass Exceptions dieser Klasse am Ende eines Signalisierungsvorgangs nicht für den Aufruf des Debuggers sorgen.

Wie in Smalltalk üblich, lässt sich das Verhalten der Exceptions bei einem Signalisierungsvorgang durch das Überschreiben der Methode für `signal` noch weiter anpassen und durch das hinzufügen von weiteren Methoden mit mehr Optionen für Handlerblöcke ausstatten.

6.2.4 Fazit

Das Exceptionsystem in Smalltalk schafft es das Signalisierungsverhalten mittels Polymorphie für unterschiedliche Fehlertypen jeweils unterschiedlich zu gestalten. Dabei ist das Interface zur Signalisierung von Exceptions mit einer einzigen Methode für `signal` äußerst simpel gehalten (im Gegensatz zu Common Lisp), da alles über die Implementation von `isResumable` und `defaultAction` geregelt wird. Auch das Problem mit rekursiven Handleraufrufen 52 ist besser gelöst als in Common Lisp, da es auch Resignalisierung an innere Handler zulässt, und man dadurch die Restarts aus Common Lisp als Subklasse von Exceptions implementieren kann.

Das Model hat allerdings auch Makel: Da die Exceptions den Signalisierungsprozess selber steuern und sich merken müssen welcher Handler gerade aktiv ist, höchst mutable. Das Ex-

ceptionobjekt ist damit also die Repräsentation des Fehlerereignisses und auch gleichzeitig die Repräsentation des Fehlersignalisierungsprozesses. Dies widerspricht damit dem "single responsibility principle"[ST04] des objektorientierten Designs und lässt die Frage aufkommen, wie sich die Zuständigkeiten trennen lassen.

Desweiteren ist es nicht immer von Vorteil, wenn eine die Fortsetzbarkeit einer Exception von ihrem Typen abhängt. Eine Methode, welche ein `ZeroDivide`-Objekt signalisiert, ist unter Umständen nicht in der Lage damit klarzukommen dass ein Handler die Ausführung fortsetzt. Um dies zu verhindern, müsste nur für diese Methode extra eine `NonResumableZeroDivide`-Subklasse mit einer überschriebenen Methode für `isREsumable` erstellen. [4]

6.3 Konzeptionelle Unterschiede

Was die beiden Fehlerbehandlungssysteme voneinander unterscheidet, sind hauptsächlich die unterschiedlichen Betrachtungswinkel zur Fehlerbehandlung:

In Common Lisp wird die Behandlung von Fehlern wird die Behandlung von Fehlern als eine Globale Aufgabe gesehen, wo es globale Schnittstellen, in der Form von Signalisierungsfunktionen und Makros, zum Interagieren mit dem Fehlerbehandlungssystem gibt. Die Fehlerobjekte sind dabei nur Daten mit denen das Conditionsystem arbeitet. Diese Betrachtungsweise erlaubt das einfachere Erzwingen von Protokollen, was sich in der guten Integration von Restarts im Debugger niederschlägt.

Smalltalk hingegen verkapselt in jedem einzelnen Fehlerobjekt ein komplettes Fehlerbehandlungssystem, welches ohne äußere Unterstützung den kompletten Fehlerbehandlungsvorgang steuert. Dadurch ist es möglich einen komplett anderen Signalisierungvorgang durch das Erstellen einer Exceptionklasse zu implementieren. Allerdings sind dadurch Exceptionobjekte nicht von ihrer konkreten Signalisierung zu trennen, während man bei Common Lisp Conditionobjekte problemlos vorallokieren und mehrfach verwenden kann.

[4] Die Situation hat Ähnlichkeiten mit den "checked exceptions" von Java: Wenn eine Methode eine `FileNotFoundException` wirft, dann zwingt sie den Aufrufer sie zu behandeln, wodurch Programmierer dazu verleitet sind für bestimmte Exceptions Varianten wie `RuntimeFileNotFoundExceptions` zu definieren. Dies wäre lösbar, wenn Java zwei verschiedene Signalisierungskeywords bereitstellen würde anstatt dies über den Typ zu handhaben.

7 Design und Implementierung eines Conditionsystems für Scala

In diesem Kapitel wird das Design und die Implementierung eines Conditionsystems beschrieben, welches versucht die Stärken der Fehlerbehandlungssystem aus Common Lisp und Smalltalk zu erhalten. Es wird in der Programmiersprache Scala implementiert, weil die bisherigen Fehlerbehandlungssysteme welche ähnlich mächtig waren alle in dynamischen und dynamisch typisierten Programmiersprachen vorkamen, aber es bisher keinen Grund gibt, warum ein solches System nicht in statisch typisierten Programmiersprachen funktionieren könnte.

Scala ist eine Programmiersprache welche die Java Virtual Machine als Zielplattform hat, das Objektsystem und Exceptionsystem von Java nutzt und größtenteils für sein, für objektorientierte Programmiersprachen sonnst unüblich, ausdrucksstarkes statisches Typsystem bekannt ist.

Da Ziel ist es ein Conditionsystem zu entwickeln, welches für die Programmiersprache Scala angepasst ist, ohne in den Fähigkeiten zur Fehlerbehandlung einbüßen zu müssen.

7.1 Designziele

First-class-ness

Conditions sollen first-class-objects sein, d.h. als Parameter und Rückgabewerte von Prozeduren verwendbar und Variablen und Feldern in Datenstrukturen zuweisbar.

Dies dient dazu, dass der Benutzer des Conditionsystems möglichst keine Sonderfälle im Umgang mit Conditions lernen muss, und sie genauso behandeln kann wie jeden anderen Wert in Scala.

Zur Realisierung wird ein Modell für Conditionobjekte gewählt welches dem von Common Lisp, wo Conditionobjekte lediglich "Nachrichtenöhne Aufrufspezifische Information wie Stacktraces sind, entspricht.

Das Smalltalkmodell ist weniger intuitiv, weil dort Fehlerobjekte über Methoden zur Signalisierung und Kontrollflussmanipulation verfügen, welche in unterschiedlichen Kontexten unterschiedliches Verhalten haben und in manchen Situationen gar nicht aufgerufen werden dürfen.

Desweiteren wären Conditionobjekte mit Methoden, welche den Kontrollfluss verändern (z.B. eine `.signal()`-Methode) vom Konzept her Continuations und wären als Objekte erster Klasse nur mittels Continuations mit unbegrenztem Geltungsbereich implementierbar (welche nicht verfügbar sind).

Offene Typhierarchie

Der Benutzer des Conditionsystems soll alle Möglichkeiten des Objekt- und Klassenmodells von Scala nutzen können um Ausnahmefälle und Fehler klassifizieren zu können.

Dies ist eine selbstverständliche Anforderung, da es unmöglich ist beim Entwurf eines Conditionsystems alle möglichen Fehler- und Ausnahmefälle, die in einem Programm vorkommen könnten, vorauszuahnen und zu berücksichtigen.

Die Conditionhierarchie des Conditionsystems wird als Obertyp einen Trait mit dem Namen `Condition`, von welchem beliebige Subklassen gebildet werden können, besitzen. Dies geht über die Erweiterbarkeit hinaus die die Exceptionhierarchie von Scala/Java anbietet, da dort der Obertyp eine konkrete Klasse ist und, aufgrund dem Fehlen von Mehrfachvererbung, nicht mit anderen parallelen Hierarchien kombinierbar ist.

Unterstützung des Scala-Typsystems

Sämtliche Conditionklassen und Traits sollen mit mit Typen parametrisierbar sein, und Handler sollen für für generische Conditiontypen, “intersection types” und “existential types” sowie anderen Features des Typsystems von Scala definierbar sein.

Diese Anforderung ist zwar vom Prinzip her in den ersten beiden enthalten, wird aber trotzdem erwähnt, da das Exceptionsystem von Scala/Java keine parametrisierten Typen und weitere Typsystem-features von Scala/Java unterstützt.

Da alle Scala-Typsystemfeatures welche über das Objektmodell der JVM hinausgehen nur während des Kompilierens existieren und zur Laufzeit ërasedßind, muss eine Repräsentation des genauesten statischen Typen jedem Conditionobjekt hinzugefügt werden. Dies soll per "Manifests" (ein Feature von Scala, welches die Typen, die bei generischen Klassen und Methoden als Typparameter übergeben werden, auch als normale Parameter zur Laufzeit übergibt.

Vereinheitlichung von Conditions und Restarts

Restarts sollen einen Untertyp von Conditions darstellen und auf dem selben Mechanismus basieren.

Dies ist aus der Überlegung hervorgegangen, dass in Common Lisp Conditions und Restarts vom Konzept her äußerst ähnlich sind (beides sind Kontrollflussmechanismen mit dynamischen Scope) aber getrennte Implementationen haben.

Die Vereinheitlichung von Conditions und Restarts wird über den Wegfall der "Condition Firewall" geschehen, wodurch es möglich sein wird von äußeren Handlern Conditionobjekte an innere Handler zu signalisieren. Restarts werden dann als Handler für signalisierte Objekte eines besonderen Zweiges der Conditionhierarchie genutzt.

Dieses Modell wurde bereits in der Programmiersprache Dylan[Sha96, Fig. 11-6] genutzt und wird um eine weitere Klasse `Situation` erweitert, die die Oberklasse aller Conditionklassen, welche keine Restarts darstellen, bildet. Dies dient dafür, dass ein Programmierer in der Lage ist alles bis auf Restarts abzufangen, da Restarts einen andern Zweck als der Rest der Conditions erfüllen.

Non-Reentrant Handler

Jeder Handler soll im gesamten Signalisierungsprozess nur ein einziges mal aufgerufen werden können. Das gilt auch dann, wenn aus einem Handler heraus eine andere Condition signalisiert wird.

Dieses Verhalten ist für einen Signalisierungsprozess der auf jeden Fall terminiert notwendig und für den Aufruf eines Defaulthandlers wenn jeder Handler die Behandlung der Condition delegiert.

Aufgrund der fehlenden "Condition Firewall" wird hierfür das Modell von Smalltalk gewählt, wo Handler vor einem Aufruf maskiert werden. Jeder Handler hat dann ein

`isMasked`-Flag welches vor dem Aufruf des Handlers gesetzt wird und dafür sorgt, dass der spezifische Handler bei der Handlersuche ignoriert werden kann.

Gewohntes Defaultverhalten

Die Syntax und Semantik des Conditionsystems sollte, in einfachen Fällen, der von bekannten `throw-try-catch`-Exceptionsystemen ähnlich sein.

Durch die hohe Verbreitung von `throw-try-catch`-Exceptionsystemen sind Entwickler auch an deren Semantik gewöhnt. Daher sollte ein neues Fehlerbehandlungssystem versuchen sie nicht zu überraschen.

Das reguläre Verlassen eines Handlerblocks wird den Kontrollfluss am lexikalischen Definitionspunkt des Handlers fortsetzen. Dies entspricht dem Verhalten der Schemeimplementation 53 aus Abschnitt 5.1.2.

Polymorphes Signalisierungsverhalten

Das Signalisierungsverhalten von Conditions sollte durch das Erstellen von Subklassen beeinflussbar, oder gar komplett ersetzbar sein.

Obwohl eine solche Flexibilität in den meisten Fällen nicht notwendig ist, garantiert die Reimplementierbarkeit der Semantik die Güte der vorhandenen Introspektionsmöglichkeiten[GKB91]. Außerdem lässt sich dadurch das Design des Fehlerbehandlungssystems für bisher unvorhergesehene Anwendungsfälle Nutzen.

Der Signalisierungsprozess wird, wie bei Smalltalk, über Hookmethoden der Conditionobjekte beeinflusst. Dazu wird es eine Methode geben welche, wie `defaultAction` in Smalltalk, im Fall, dass keine passenden Handler gefunden werden konnten, aufgerufen wird. Für den kompletten Signalisierungsvorgang ist eine weitere Hookmethode vorgesehen, welche dann eigene Signalisierungsprozesse erlaubt. Desweiteren sind die Defaultimplementationen der Hookmethoden nur über das öffentliche Introspektionsprotokoll realisiert.

Introspektion

Das Conditionsystem sollte grundlegenden reflektiven Zugriff auf sich erlauben. Dazu gehört der Zugriff auf Handler ohne sie auszuführen und der Zugriff auf beliebige Fehlercontinuations um von denen aus eine Condition zu signalisieren zu können.

Dies ist für eine eventuelle Implementation eines Debuggers wie in Common Lisp notwendig, sowie für das sichere Nutzen der Restarts. Zusätzlich muss das Conditionsystem Introspektion bieten um eine mögliche Reimplementation des Signalisierungsprozesses zu ermöglichen.

Zur Introspektion wird typsicherer Zugriff auf die aktuelle Fehlercontinuation, alle ihr zugehörigen Handler und alle vorherigen Fehlercontinuations gewährt. Desweiteren werden Conveniencemethoden angeboten zum Suchen von Handlern für angegebene Conditiontypen und um auf das Vorhandensein weiterer Handler für vorliegende Conditionobjekte zu prüfen. Die Introspektion wird allerdings bis auf das Ausführen von Handlern nur einen Lesezugriff erlauben.

Konsolidierung von Signalisierungsmethoden

Es sollen sich möglichst viele verschiedene Anwendungsfälle, Signalisierungs- und Resignalisierungmuster mit möglichst wenigen Signalisierungsmethoden ausdrücken lassen.

Dies soll das die Komplexität des Signalisierungsprotokolls reduzieren und sich möglichst dem Idealfall von `throw-try-catch`-Systemen annähern, welche nur überein einziges Keyword verfügen.

Es wird exakt zwei Signalisierungsmethoden geben, welche jedoch nicht auf Conditionobjekten, sondern auf Fehlercontinuations implementiert sind. Die konkrete Fehlercontinuation wird angeben welche Handler sichtbar sind und die ausgewählte Methode auf der Fehlercontinuation angeben ob die Berechnung an der Stelle fortgesetzt werden darf oder nicht.

7.2 Schnittstellen

7.2.1 Handlerdeklaration

Die Deklaration von Handlern ist mittels einer ad-hoc-Syntax in Scala möglich. Mittels eines Compilerplugins oder eine Präprozessors könnte sie weitaus verbessert werden:

```
// single handler
block {
  // protected block
} guard from { (c : Error, here : Frame) =>
  // handler
}

// multiple handlers
block {
  // protected block
} guard List( from { (c : Warning, here : Frame) =>
  // handler 1
}, from("handler hame") { (c : Error, here : Frame) =>
  // handler 2
})
```

Codebeispiel 71: Handlersyntax

Dabei ist der erste Parameter die signalisierte Condition und der zweite Parameter (per Konvention `here` genannt) stellt die Fehlercontinuation des geschützten Blocks dar. Durch die Nutzung der `here`-Fehlercontinuation ist es möglich Conditions zu resignalisieren und weiter innen liegende Handler dabei zu ignorieren.

Zu beachten ist hierbei, dass Handler optional mit Strings benannt werden, was zum Benennen von Restarts verwendet werden kann.

7.2.2 Fehlercontinuations

Der Typ der Fehlercontinuations heißt `Frame` da dies einen relativ Kurzen und teilweise passenden Namen darstellt. Alternativen wie `Context`, oder `SignalContext` wurden auch in Erwägung gezogen, aber verworfen, da der Typ jedes mal in einer Handlerdeklaration angegeben werden muss.

Die Reifikation der aktuellen Fehlercontinuation (ab hier: `Frame`) erhält man über das Singletonobjekt `Frame` und dessen Methode `currentFrame`.

```scala
sealed trait Frame {
  def isAlive : Boolean
  def raise(c : Typed[Condition]) : Nothing
  def signal(c : Typed[Condition]) : Unit
  def fromHere[Ret](thunk : => Ret) : Ret
  def apply[Ret](thunk : => Ret) : Ret
}
```

Codebeispiel 72: `Frame`-trait: Schnittstelle für Fehlercontinuations

Die `isAlive`-Methode gibt an, ob der `Frame` noch aktiv ist. Sie liefert `false`, wenn sie aus einem anderen Thread heraus aufgerufen wird oder die Fehlercontinuation ungültig geworden ist. Aufrufe von anderen Methoden haben undefiniertes Verhalten wenn `isAlive` false zurückgibt.

`raise` ist eine Signalisierungsmethode, welche ein Conditionobjekt erwartet, und es signalisiert ohne zurückkehren zu können. Es entspricht somit der `error`-Funktion aus Common Lisp.

`signal` ist eine Signalisierungsmethode, welche ein Conditionobjekt erwartet, einen `SignalResume[c.type]`-Restart für dieses spezifische Conditionobjekt[1] anlegt und d nn `raise` aufruft. Das bedeutet, dass durch das Signalisieren einer `SignalResume[c.type]`-Condition vom exakt selben Typ (also vom selben Objekt) die Berechnung wieder beim Aufrufer von `signal` fortgesetzt wird. Dies entspricht ungefähr `cerror` aus Common Lisp und `signal` aus Smalltalk.

`fromHere` und `apply` erwarten beide einen Block als by-name-Parameter und führen ihn in einem Scope aus, wo `Frame.currentFrame` auf den Reciever verweist. Dies erlaubt das ausführen von Code unter einer “Condition Firewall” wo nur die Handler sichtbar sind die zum Reifikationszeitpunkt der Fehlercontinuation sichtbar waren.

Alle diese Methoden lassen sich auch auf dem globalen Singletonobjekt `Frame` ausführen, welches sie an die gerade aktive Fehlercontinuation delegiert.

[1] `obj.type` ist Syntax für einen Singletontyp. D.h. nur dieses eine spezifische Objekt ist Instanz dieses Typs.

7.2.3 Signalisierungskombinationen

Da man innerhalb eines Handlers Zugriff auf sowohl die innerste Fehlercontinuation (mittels `Frame.currentFrame` oder einfach nur `Frame`) als auch an die die dem Handler als Parameter übergeben wurde (idiomatisch `here`), und jede der Continuations über zwei Methoden zur Signalisierung verfügt, hat man vier verschiedene Möglichkeiten eine Condition zu signalisieren:

`here.raise(c)` signalisiert an weiter außen liegende Handler und kehrt nicht mehr zurück. Dies entspricht `pass` aus Smalltalk.

`here.signal(c)` signalisiert an weiter außen liegende Handler und kehrt bei einer `SignalResume[c.type]` Condition wieder zurück. Entspricht `outer` aus Smalltalk.

`Frame.raise(c)` signalisiert auch an innere Handler und kehrt nicht zurück. Entspricht `resignalAs:` in Smalltalk.

`Frame.signal(c)` signalisiert auch an innere Handler und kehrt bei der nächsten `SignalResume[c.type]`-Condition wieder zurück. Das stellte sich als nützlich heraus um Restars auszuführen, bei denen man sich nicht sicher ist ob sie angelegt wurden.

Grundsätzlich bedeutet `raise` immer "es ist eine Situation aufgetreten, mit der ich nicht mehr klarkomme" und `signal` "ich weiß wie ich mit dieser Situation klarkomme, aber vielleicht wissen es andere besser", während `here` zur Resignalisierung von Conditions und `Frame` für Restarts oder Fehlerbehandlungsstrategien benutzt werden soll.

7.2.4 Introspektion von Fehlercontinuations

Da, im gegensatz zu Common Lisp, die Notwendigkeit zur Benutzung von Introspektion für die Fehlerbehandlung gemindert wurde [77], sind die reflektiven Operationen nur durch den Import von `reflective.frameOperations` sichtbar. Der Import stellt eine weitere Gruppe von Methoden auf Fehlercontinuationobjekten des Traits `Frame` bereit:

```scala
def handlersFor[C <: Condition : Manifest] : Seq[Handler[C]]
def handlersFor(c : Typed[Condition]) : Seq[Handler[c.type]]
def findHandler[C <: Condition : Manifest] : Option[Handler[C]]
def findHandler(c : Typed[Condition]) : Option[Handler[c.type]]
def isHandlerDefined[C <: Condition : Manifest] : Boolean
def isHandlerDefined(c : Typed[Condition]) : Boolean

def ownHandlers : Seq[Handler[_]] = f.handlers
def nextFrame : Frame = f.next
def allFrames : Seq[Frame] = f.frameIterator.toSeq
```

Codebeispiel 73: Reflektive Schnitstelle für Fehlercontinuations

Die ersten sechs Methoden stellen abstrakte Operationen dar, welche vom Nutzercode verwendet werden sollen, während die letzten drei zur Reimplementation des Signalisierungsvorgangs dienen.

Die abstrakten Abfragemethoden existieren in jeweils 2 Varianten: Eine für den Fall dass man kein Conditionobjekt zur Verfügung hat, welche dann mit einem Typen aufgerufen werden kann (z.B.`Frame.handlersFor[DepricationWarning[ComponentA]]`). Die anderen hingegen erwarten ein konkretes Conditionobjekt, zu dem dann Handler gesucht werden. Diese Methoden sind alle type-safe und verbieten den Aufruf von Handlern mit einem inkompatiblen Conditionobjekt. [2] Dabei ist zu beachten, dass die Methoden nicht überprüfen ob Handler maskiert sind oder nicht.

Die übrigen drei Methoden erlauben den direkten Zugriff auf die verkettete Struktur der Fehlercontinuations und deren Handler, wobei Handler in diesem Fall einen Wildcardtypen als Typparameter haben, und deshalb nicht direkt aufrufbar sind.

[2] Die Methoden welche ein konkretes Conditionobjekt erwarten, benötigen ein Typsystemfeature (dependant singleton types) von Scala welches noch experimentell (`-Xexperimental`) ist. Zur Typsicherheit müssen die Methoden den Singletontyp des übergebenen Conditionobjektes als Rückgabetyp enthalten.

7.2.5 Introspektion von Handlern

```
sealed trait Handler[-C <: Condition] extends Function[C, Nothing] {
  def name : String
  def requiredType : Manifest[_ <: Condition]
  def frame : Frame

  def isMasked : Boolean
  def isAlive : Boolean

  def applicableFor[T : Manifest] : Boolean
  def applicableFor(c : Typed[Condition]) : Boolean
  def apply(condition : C) : Nothing

  def tryCast[TryC <: Condition : Manifest] : Option[Handler[TryC]]
  def tryCast(c : Typed[Condition]) : Option[Handler[c.type]]
}
```

Codebeispiel 74: Interface für Handlerfunktionen

Handler sind als Subtyp von Funktionen implementiert, da sie tatsächlich Funktionen sind.

Sie haben drei einfache Zugriffsmethoden `name`, `requiredType` und `frame`, welche jeweils ihren Namen (oder null), eine Repräsentation des Conditiontyps welchen sie behandeln und die Fehlercontinuation welche bei ihrer Definition aktiv war, zurückgeben.

Die nächsten zwei Methoden, `isMasked` und `isAlive`, dienen zur Abfrage ob der Handler bereits Teil eines laufenden Signalisierungsprozesses ist oder zur Abfrage ob der Handler noch Teil einer gültigen Fehlercontinuation ist. Das Aufrufen eines ungültigen Handlers (oder eines Handlers aus einem fremden Thread) hat hierbei die Signalisierung einer (nicht fortsetzbaren) `DeadHandlerInvocation` zur Folge.

Die Methoden `applicableFor` und `apply` dienen zur Typprüfung und zum Aufrufen des Handlers, wobei die Typprüfung sowohl gegen Typen oder ein Conditionobjekt erfolgen kann.

Die letzten zwei Methoden dienen zum typsicheren Casten der Condition zu einem bestimmten Typ. Dabei gibt der Handler eine in einem Optionstypen verpackte Referenz auf sich selber zurück.

Zu beachten ist, dass Handler beim Aufruf nicht prüfen ob sie maskiert sind oder nicht, weil die Signalisierungsstrategie Teil des Exceptionobjektes ist. Der Aufruf von Handlern mittels dieser Introspektion kann also Endlosrekursionen zur Folge haben.

7.2.6 Hookmethoden in der Conditionhierarchie

Die Oberklasse aller Conditions `Condition` bietet drei überschreibbare Hookmethoden zum Beeinflussen des Signalisierungsprozesses, welche in der Wurzelklasse folgende Defaultimplementation haben:

```
protected def noHandlerFound : Nothing =
  Frame.raise(new SignalResume[this.type])

protected def doRaise(frame : Frame) : Nothing = {
  frame.handlersFor(this).find(!_.isMasked) match {
    case Some(handler) => handler(this)
    case None => noHandlerFound
  }
}

protected def doSignal(frame : Frame) : Unit = block {
  doRaise(frame)
} guard from { (re : SignalResume[this.type], _ : Frame) => () }
```

Codebeispiel 75: Hookmethoden in `Condition`

Diese drei Methoden sind für den gesamten Signalisierungsprozess verantwortlich und, wobei die erste(`noHandlerFound`) der Methode `defaultAction` aus dem Smalltalk-Exceptionsystem entspricht.

Die anderen beiden Methoden werden von `Frame#signal` und `Frame#raise` aufgerufen und zeigen, wie hier das Wiederfortsetzung einer Berechnung mittels Eines Restarts geschieht.

Es gibt zwei Fälle wo die Hookmethode `noHandlerFound` überschrieben wird: Einmal in einer Subklasse namens `Warning`, wo die Condition auf `Console.out` ausgegeben wird und einmal in einem Mixin-Trait `Serious`, wo eine weitere Condition `NoHandlerFoundError` signalisiert wird, deren `noHandlerFound`-Methode die Berechnung endgültig mittels einer Java-Exception abbricht.

Die Methode `doSignal` und `doRaise` werden hingegen in keiner Subklasse überschrieben, könnten aber, dank der äußerst banalen Implementation, problemlos mit eigenem Signalisierungsmethoden überschrieben werden.

7.3 Anwendungsfälle

Das vorgestellte Conditionsystem bietet vorallem in einigen Gebieten nützliche Fortschritte an:

Erstens setzt es das Typsystem von Scala fast vollständig um. Die bedeutet, dass Conditions über Typparameter verfügen können und auch von Handlern für parametrisierbare Typen abgefangen werden können. Dabei werden intersection-, singleton-, bounded- und existential-types unter Berücksichtigung co- und contravarianter Typparameter unterstützt.

Vor allem die Unterstützung von Typparametern kann es erlauben sehr präzise Handler zu definieren und zu signalisieren.

```
block {
  IOLib.readAndDoSomethingWithFile(file)
} guard from { (c : FileError, here : Frame) =>
  here.raise(new TechnicalError
                 with Cause[FileError]{def cause = c}
                 with Subsystem[IOLibrary]{})
}
// somewhere else
block {
  //...
} guard from { (c : Subsystem[_ <: OSLayer], here : Frame) =>
  //...
}
```

Codebeispiel 76: Parametrisierte Typen zur Klassifizierung von Fehlern

In diesem Beispiel wurde gezeigt, wie durch die Verwendung von Mehrfachvererbung und eines Typparameters welcher gegen einen Wildcardtypen gematched wird, komplexe Fallunterscheidungen, die normalerweise notwendig Wären, durch das Typsystem von Scala, vermieden werden.

Ein praktisches Beispiel für die Verwendung von Typparametern ist die Implementierung von typsicheren Restarts:

```scala
trait Associated[+C <: Condition] extends Condition
trait Continue[+T] extends Restart {
  def returnValue : T
}

def div(a : Int, b : Int) = block {
  if(b != 0) a / b else Frame.raise(new DivZeroErr(a){})
} guard from {(c : Continue[Int]
                  with Associated[DivZeroErr], _ : Frame) =>
  c.returnValue
}
block {
  Seq(1,2,3,4,0,5).map(div(20,_))
} guard from {(c : DivZeroErr, here : Frame) =>
  Frame.signal(new Continue[Double]{def returnValue = Double.MaxValue}
                  with Associated[DivZeroErr]) // try this
  Frame.signal(new Continue[Int]{def returnValue = Int.MaxValue}
                  with Associated[DivZeroErr]) // try this
  here.raise(c) // resignal
}
//res: Seq[Int] = List(20, 10, 6, 5, 0, 4)
```

Codebeispiel 77: Implementierung von typsicheren Restarts

In diesem Beispiel sieht man dass Restarts mit Rückgabewerten parametrisierbare Conditiontypen [3] benötigen um typsicher zu sein.

Eine weiteres nützliches Ergebnis was sich, durch das Design dieses Conditionsystems, herausgestellt hat ist die Verwendung der, auf den ersten Blick unnützen, Semantik von `Frame.signal()` zur abgesicherten Ausführung von Restarts ohne, wie bei Common Lisp, sich der Introspektion zu bedienen. Dabei wird ausgenutzt, dass Restart die `noHandlerFound`-Methode nicht überschreiben und deshalb automatisch zum letzten `signal`-Aufruf zurückkehren.

[3] oder sehr viele redundante Resumeklassen (`ContinumeInt`, `ContinumeString`, `ContinumePerson` ...)

8 Fazit

Der Überblick über die Fehlerbehandlungsmittel und -systeme, die im Bereich der Programmiersprachen angeboten werden, zeigt, dass ein Großteil der bisherigen Systeme von Unzulänglichkeiten geplagt ist. Dabei erlauben diese Systeme dem Programmierer nicht die Fehler auf abstrakte Art und Weise zu behandeln. Vor allem die anschließende Betrachtung der Fehlerbehandlunssysteme von Common Lisp und Smalltalk, zeigt deutlich die Vorteile auf, die sich ergeben, wenn sowohl passende Mittel zur Fehlersignalisierung, Fehlerklassifizierung als auch Fehlerbehandlung zur Verfügung gestellt und genutzt werden.

Durch die Implementation eines simplen Fehlerbehandlungsprototypen in Scheme wurde ein Überblick verschafft über Implementationsstrategien und -tücken eines Systems welches in der Lage ist, die in Abschnitt 2.2 genannten, Anforderungen umzusetzen. Die anschließende Analyse der konkret existierenden Fehlerbehandlungssysteme in Common Lisp und Smalltalk war Motivation und Designziele für die Entwicklung eines neuen Fehlerbehandlungssystems in Scala geliefert.

Dieses Fehlerbehandlungssystem in Scala hat weitere Erkenntnisse hervorgebracht. Dass die Implementation eines solchen Systems in einer statisch typisierten Programmiersprache, ist nicht nur möglich ist, sondern dass sie weitere Möglichkeiten zur Fehlerklassifizierung und -behandlung erlaubt.

Somit lässt sich sagen, dass die Möglichkeiten die existieren, um bestehende Werkzeuge zur Fehlerbehandlung zu verbessern und weiter auszubauen, längst nicht erschöpft sind und Designer vor Programmiersprachen dazu ermutigt werden sollten, Fehlerbehandlung jenseits von üblichen `throw-try-catch`-Exceptionsystemen anzubieten.

Literaturverzeichnis

[APBP10] Oscar Nierstrasz Andrew P. Black, Stéphane Ducasse and Damien Pollet. Pharo by Example 2. draft chapter: Handling exceptions, 2010. https://gforge.inria.fr/frs/download.php/26600/PBE2-Exceptions-2010-03-02.pdf.

[Fee03] Marc Feeley. SRFI 39: Parameter objects, 2003. http://srfi.schemers.org/srfi-39/srfi-39.html.

[GKB91] Jim des Rivieres Gregor Kiczales and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[HAS96] Gerald Jay Sussman Harold Abelson and Julie Sussman. *Structure and Interpretation of Computer Programs (SICP), Second edition*. MIT Press, 1996.

[Hoa09] C. A. R. Hoare. Null references: The billion dollar mistake. *Presentation at QCon London*, 2009. http://qconlondon.com/london-2009/presentation/Null+References:+The+Billion+Dollar+Mistake.

[KCR+98] R. Kelsey, W. Clinger, J. Rees, et al. Revised^5 Report on the Algorithmic Language Scheme (R5RS). In *ACM SIGPLAN Notices*, 1998.

[Mic11] Microsoft Corporation. *[MS-ERREF]: Windows Error Codes*, 2011. http://msdn.microsoft.com/en-us/library/cc231196(v=prot.10).aspx.

[Moo74] David A. Moon. *MACLISP Reference Manual, Revision 0*. MIT, 1974.

[Pc89] Kent Pitman and The X3J13 committee. CLOS-CONDITIONS:INTEGRATE, 1989. Standardization proposal.

[Pit90] Kent Pitman. Exceptional situations in Lisp. In *First European Conference on the Practical Application of Lisp (EUROPAL'90)*, 1990.

[Pit01] Kent Pitman. Condition handling in the Lisp language family. *Advances in exception handling techniques*, pages 39–59, 2001.

[SG96] Guy Steele and Richard P. Gabriel. The evolution of Lisp. In *History of programming languages, Volume 2*, pages 233–330. ACM, 1996.

[Sha96] A. Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley, 1996. http://www.opendylan.org/books/drm/Title.

[ST04] A. Shalloway and J. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley, 2004.

[Ste90] Guy Steele. *Common LISP: the language*. Digital Press, 1990.

[The04] The IEEE and The Open Group. *The Open Group Base Specifications Issue 6 – IEEE Std 1003.1, 2004 Edition*. IEEE, 2004. http://www.opengroup.org/onlinepubs/009695399/.

[WG199] WG14. ISO C Standard 1999. Technical report, ISO, 1999. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

[Woo97] B. Woolf. The null object pattern. In *Pattern languages of program design 3*, pages 5–18. Addison-Wesley Longman Publishing Co., Inc., 1997.

Liste der Codebeispiele

1 Funktionsprototyp von `atoi` . . . 6
2 Funktionsprototyp von `strtol` [1] . . . 7
3 Funktionsprototyp von `fgetc` . . . 8
4 Aufruf von `fgetc` . . . 9
5 Interaktion mit `string->number` . . . 9
6 Interface für `java.util.Map` . . . 10
7 Das Nullobjekt-Pattern in Java . . . 11
8 Spezialisierung eines Protokolls auf `nil` . . . 12
9 Aufruf der polymorphen Funktion . . . 13
10 Hashtable in Common Lisp . . . 14
11 Interaktion mit einer Hashtable in Common Lisp . . . 14
12 Funktionsprototyp von `pthread_create` . . . 15
13 Ein Aufruf von `pthread_create` . . . 16
14 `out`-Parameter in C# . . . 16
15 Benutzung von `errno` . . . 18
16 `option`-Typdefinition in Standard ML . . . 19
17 Interaktion mit `Int.fromString` . . . 19
18 Extraktion eines `int` aus `int option` . . . 19
19 Verwendung von `Maybe` in `do`-Notation . . . 20
20 Signalisierung über Rückgabewerte . . . 21
21 Fallunterscheidung am Rückgabewert . . . 21
22 `string->integer` mit einem Errorhandler als Parameter . . . 22
23 Aufruf mit einem Errorhandler . . . 22
24 Errorhandler in Smalltalkdialekten . . . 22
25 Wiederverwendung eines Errorhandlers mittels `let` . . . 23
26 Verkettete Errorhandler . . . 23
27 Interaktion mit `old-enough?` . . . 24

28 Implemantation von `string->integer` im Continuation-Passing-Style 24
29 Aufruf mit expliziten Continuations 25
30 Abbruch von Berechnungen mittels einer Continuation 26
31 Verkettung von Fehlercontinuations 27
32 Interaktion mit `mature-musicians?` 28
33 Definition eines Parameterobjektes für Fehlercontinuations 28
34 Definition von `throw` und `extend-failure-continuation` 29
35 Definition einer `try-catch`-Syntax 29
36 Nutzung der `try-catch`-Syntax 30

37 Deklaration einer Exception in Ada 34
38 Fallunterscheidung bei Exceptions in Ada 35
39 Syntax zur Falluterscheidung von Fehlern in Java 36
40 Fallunterscheidung von Fehlern in JavaScript 37
41 Reinterpretation durch Typmutation 39
42 Reinterpretation durch Verkettung 40
43 Fallunterscheidung an verketteten Fehlern 41

44 Parsefunktion mit `throw` als Fehlersignalisierungmethode 44
45 Parsen von Ganzzahlen aus einer Datei 44
46 `parse-integer-file` welches Konvertierungsfehler ignoriert 45
47 Verschiedene Fehlerbehandlungsstrategien 46
48 Signalisierung eines Fehlers inklusive Behandlungsstrategien 47
49 Nutzung von eingebetteten Fehlerbehandlungsstrategien 48
50 Definition von `extend-failure-continuation` aus Abschnitt 3.2.3 49
51 Redefinition von `extend-failure-continuation` ohne `failure-k` 50
52 `try-catch` mit redefinierter `extend-failure-continuation` 50
53 Endgültige Definition von `extend-failure-continuation` 51
54 Fehlerbehandlung auf Basis von Escapecontinuations 51

55 Aufruf von Signalisierungsfunktionen 54
56 Nutzung von `handler-case` 56
57 Nutzung von `handler-bind` 56
58 Die “Condition Firewall” von Common Lisp 57
59 Definition einer Conditionklasse 58
60 Interaktion mit Restarts im Debugger 59

61 Aufruf von `invoke-restart` . . . 60
62 Definition von `parse-integer-file` in Common Lisp . . . 61
63 Interaktion mit `parse-integer-file` . . . 62
64 Exceptionhandler in Smalltalk . . . 64
65 Fehlersignalisierung per Reflection . . . 65
66 Resignalisierung in Smalltalk . . . 66
67 `retry` und `retryUsing:` . . . 67
68 Wiederaufnahme einer Berechnung . . . 68
69 `resume:` von `MessageNotUnderstood` . . . 69
70 Semantik von `outer` . . . 70

71 Handlersyntax . . . 78
72 `Frame`-trait: Schnittstelle für Fehlercontinuations . . . 79
73 Reflektive Schnitstelle für Fehlercontinuations . . . 81
74 Interface für Handlerfunktionen . . . 82
75 Hookmethoden in `Condition` . . . 83
76 Parametrisierte Typen zur Klassifizierung von Fehlern . . . 85
77 Implementierung von typsicheren Restarts . . . 86

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 6. Juli 2011 Alexander Konstantinov