TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Extending Clean

with Exception Handling

By

S. van den Berg

Supervisors:

prof.dr.ir. M.J. (Rinus) Plasmeijer (RB)
drs. Betsy Pepels (RB)
prof.dr. Bruce W. Watson (TU/e)

Eindhoven, June 2005

Abstract

This thesis describes the current state of exceptions in lazy functional languages. A taxonomy of exception handling is presented. The need for exceptions from a software engineering viewpoint is discussed. Finally a proposal for a new type of exceptions and their handling is presented. The target language of this thesis is the pure, lazy functional language Clean.

Contents

Listings

Chapter 1

Prelude

This document was written in order to fulfill the requirements for the master's grade in computer science. The idea sprang from the "Wish list" from Marco Kesseler. This is a list that is distributed on the Clean mailing list. This list describes the features users would like to see in the language Clean. One of these features is exception handling. Over time, exceptions became a bigger priority in Clean, due to the development of an operating system in Clean. An ad-hoc implementation of Haskell-like exceptions was created. In order to improve the way errors are handled in large programming projects this research continued in the form of my master's research project. My personal reasons to take on this project were my fascination with functional languages. I like the clean and elegant way of expression. Also, the close link between functional languages and mathematics intrigues me. As a practicing programmer I can appreciate the clean syntax of most functional languages. One of the goals of this research was, for me personally, to make functional languages more suitable to real world application, through better error handling. Exceptions in lazy functional languages is one of the most underestimated topics in programming language design and implementation. As you will see in this document, it has many intricate details and has repercussions for almost everything in language design. From personal experience I know this wasn't an easy project. It is hard to keep track of all the effects a design has on other parts of the language.

Chapter 2

Introduction

To fulfill the requirements for the degree of Master of Science at the Eindhoven University of Technology, this document was written. The project took place at the Radbout University of Nijmegen, under supervision of Rinus Plasmeijer and Betsy Pepels. Research started in September 2002 and ended in April 2005.

In this thesis we investigate the design and implementation of exceptions and exception handling in the lazy functional language Clean. Exceptions are a feature of functional programming languages that don't receive a lot of research exposure. Until now only one implementation of exception handling in a lazy functional language exists. Exceptions in imperative languages en strict functional languages are traditionally implemented as changes in execution order. Lazy functional programming languages do not allow this.

The structure of this document follows from the structure and order of the research. At first we will try to investigate in detail the problems that arise when one wants to add exception handling to a language. We start by looking at the techniques used in imperative languages (Chapter 4), as exception handling is most commonly used in imperative languages. Second, we will investigate exception handling in a class of languages closer to lazy functional languages: strict functional languages (Chapter 5). These chapters show us examples of the practical use of exception handling in other languages.

The following chapter gives a detailed account of the problems of implementing exception handling in lazy, pure functional languages (Chapter 6). The detailed description of the problems together with the demands stated in the problem description lead to a number of demands to evaluate existing solutions. These solutions are evaluated in Chapter 7.

After the discussion of problems in lazy functional languages and exceptions, we describe why we want to go trough the trouble of changing a language to add exceptions and exception handling. Why are exceptions so important? (Chapter 8)

Now that we have stated our demands and evaluated and pinpointed the problems of existing exception handling techniques, we introduce our alternative design for exceptions in Clean (Chapter 9). The next chapter describes an implementation of the proposed design as a prototype (Chapter 10). These two chapters are the central chapters of this thesis.

Following the design and prototype descriptions, we discuss the effects of our design, compare it to haskell's and point out a number of interesting issues (Chapter 11). The last chapter, the conclusion, describes the lessons learned and possible directions for future research (Chapter 12).

Chapter 3

Problem statement

Recent research[vWP03] on operating systems written in Clean renewed the call for a clean exception handling mechanism in the lazy, pure functional language Clean[NSvEP91]. Current implementations in other lazy functional languages do not provide a nice clean design coupled with good programmer usability.

Our focus will lie on the software engineering side of exception handling. We will primarily search for a mechanism that's practical in it's use. The approach is driven by practical applications and use in large scale software products.

In our research project, we examine alternative designs for exception handling in lazy evaluated, strongly typed functional languages. Alternative designs should have the following properties:

1. **Usable design** The design should not place a big burden on programmers. Exception handling should be easy to use and should not require a large number of adaptations on source code.

2. **Efficient implementation** Implementation should not, or only in a minor way, affect evaluation efficiency.

3. **Intuitively clear semantics** It should be clear how programmers should use the exception handling mechanism. Furthermore the effects of using exception handling should be clear without having to read formal semantics.

4. **Semantically correct** The exception handling mechanism semantics should be formally correct. This implies the semantics should be deterministic and non-ambiguous. We want to show such a semantics

exists, not deliver a complete and non-ambiguous operational semantics.

We restrict ourselves to synchronous exceptions, as they are the most used by programmers. This thesis develops an alternative exception handling mechanism and compares the alternative mechanism to existing mechanisms.

Chapter 4

Exception handling in imperative languages

We explore the use of exceptions in imperative programming languages in this chapter. In contrast to imperative (OO) languages, exception handling in functional languages is still relatively young. There are many other possibilities to handle errors in functional languages, therefore exceptions are not that often used. In imperative languages though, the use of exceptions is very common. These languages are usually object oriented. This is why we look into exception handling in imperative languages to find typical patterns of usage.

4.1 Examples of exception handling syntax and semantics

In our research on exception handling in imperative languages we will use three representatives of the class of languages. The languages we choose are: Java[AGH00], C++[Str91] and C#[HWG03]. We chose these languages due to the fact that they are more-or-less statically typed, like Clean is, and are often used for complex information systems that require good handling of errors. Furthermore, all languages are relatively modern and support established exception handling methods and keywords.

4.1.1 throw and catch

Most languages that support exception handling support at least two syntactical indicators for exceptions. These are **throw** to indicate the raising,

or occurence of an error or exception and **catch** to indicate the code block or statements that process the raised exception or error.

In Java we have the keywords **try** and **catch** to indicate the use of exceptions. A **try {}** block indicates that the code between { and } is coupled to the exception handler defined in the **catch** code block. In C# the syntax is the same as in Java.

C++ is different from Java and C# with respect to the type of the exception. In C++ any object can be returned as an exception. Exceptions are first class objects. C++ uses **throw** to raise an exception and **catch** to handle exceptions. In Java as well as in C# exceptions are forced to either implement an interface or subclass an Exception class.

4.1.2 Matching on exceptions

In Java and C# exceptions are matched by type. For example the following code (listing 4.1), shows a type match in the **catch** statement.

Listing 4.1: Java's exception matching

```
try {
 //a method that throws an exception,
 //defined by throws x;
 Database db = someClass.getDatabase(databaseName);
} catch ( x e ) {
 //handle exception x thrown by the database class
 e.printStackTrace();
} catch ( Exception ex) {
 //falltrough handler, handles all exceptions
 ex.printStackTrace();
} finally {
 //close or do other things with failed database
 //connector
}
```

Here the exception with type x, supertype *Exception* or implementation of *Throwable* is matched by the first **catch**. The second **catch** matches the supertype. In C# a **catch** without arguments is used as a catch all statement.

In C++ matching is done on the type of what was thrown. For example if within a **try...catch** block a **throw** is issued with type E, a **catch(E)** will match. For precise semantics we refer to the compagnion paper, "A comparison of exception-handling mechanisms"[vdB03].

checked exceptions

The language designers of Java made an interesting choice in handling runtime exceptions. They implemented **checked exceptions** in Java. Checked exceptions mean, exceptions that are thrown must be declared by a **throws** keyword. Methods that call other methods that declare a **throws**, MUST handle the exception. These methods must either re-throw the exception or declare a **throws** themselves. Checked exceptions are a technique to check at compile time if exceptions are handled well.

This however gives problems with asynchronous, runtime exceptions. Runtime exceptions are not declared and can be raised anywhere in the code. Java defined an interface called RuntimeException, implementations of this interface contain various exceptions occurring at runtime like for example IndexOutOfBoundsException.

This design gives rise to a number of problems. First of all RuntimeException is a subinterface of Exception. If a programmer uses a **catch(Exception e)**, the **catch** also catches runtime exceptions! Furthermore one cannot use versioning on methods. If one has a method that throws exception A and B, and later on the programmer wants to add an exception C, all client code needs to be rewritten to satisfy the checked exception constraints. A number of authors have noticed the problems with checked exceptions, notably Anders Hejlsberg, head designer of C#.

Other languages like C#, Python or C++ do not force you to check the exceptions. This allows for a more flexible error handling, though it introduces the possibility to ignore an exception.

4.1.3 finally constructions

Another technique that is often used in imperative programming languages is the use of a **finally** statement. The semantics for **finally** are: a block of code contained in a **try { ... } catch { } finally { }** is executed until it either finishes or an exception is raised. In case of an exception that matches the **catch** clause, the exception handler is executed, after which the code in the **finally** block is executed. In all other situations the code in the **finally** block is executed last.

finally codeblocks are often used for cleaning up after errors or other administrative tasks.

4.2 Examples of usage

This paragraph shows several examples of problems commonly solved with exceptions in imperative languages. Typical applications are the abstraction of errors in API's trough exception wrapping and rethrowing. Other often used techniques are environment cleanup and printing debugging statements.

4.2.1 Error abstraction through wrapping

A common strategy to keep API's modular is to abstract errors occurring within the library to a more general exception. An example is wrapping and SQLException in a data-storage layer to a RecordNotFound or similar exception. This technique can be implemented in several ways: using checked exceptions like java's to only force certain exceptions to be checked. Another technique is to wrap the exception to another exception. See listing 4.2.

Listing 4.2: Wrapping exceptions

```
//C#: Exception Handling: Handling all exceptions
using System;
class MyClass {
 public void Method() {
   try {
         int x = 0;
         int sum = 100/x;
    } catch(DivideByZeroException e) {
           //throw a new exception, wrapping the older
           //one, possibly adding information.
        throw new MethodFailureException(e);
    }
  }
}
class MyClient {
  public static void Main() {
    MyClass mc = new MyClass();
   try {
       mc.Method();
    } catch(Exception e) {
      Console.WriteLine("Exception caught here");
    }
  Console.WriteLine("LAST STATEMENT");
```

```
  }
}
```

4.2.2 Safety net constructions

Another often used application of exceptions is cleaning up or repairing after an error occurred. Examples are: trying a new database connection if another one failed, cleaning up a dataset after an incorrect SQL statement etc. The following listing (listing 4.3) shows cleaning up after an error.

Listing 4.3: Recovering after errors

```
public class MyClass {
  public Record getRecord() {
    Database database = null;
    Record record = null;
    try {
       database = getDatabase();
       record = database.getRecord(
          "SELECT rec FROM sometable WHERE id=1");
        return record;
    } catch (SQLException e) {
        log.error(
           "Record not found or database unavailable");
    } finally {
        //check if the database failed or the retrieval
        if ( database == null ) {
            throw new DataLayerCriticalError();
        }
        if ( record = null ) {
            //return a new empty record
            return new Record();
        }
    }
  }
}
```

Instead of throwing a wrapped exception in listing 4.3 we could also have re-tried getting another database connection.

4.3 Implementation details

Exception handling in imperative languages is often implemented using stack unwinding. Most languages are stack based. The stack contains pointers to local constructs used in methods or functions. An error handler is installed on the stack as a pointer to a method that handles an exception. These handler frames are installed for example upon entry of a method. If an exception is thrown, the stack is unwounded frame by frame from top to the handler frame. This unwinding cleans up any objects allocated during the method. This stack based approach guarantees nested exceptions and handlers work properly.

Many operating systems support exception handling in a matter described above. For example, Windows defines a **struct** to register an exception handler frame (EXCEPTION_REGISTRATION). Often, basic exceptions like divide-by-zero and nullpointer exceptions are predefined by an operating system. Other functionality can be methods to denote a stack-frame as an exception handler.

4.4 Type system details

Exceptions form a challenge to type systems. Exceptions are part of the return values of each function that raises an exception. Therefore exceptions need to be in the type that each and every function returns. This can only be the case if the language supports subtyping. C++ as well as Java support this. The solution is to add the type Exception to the minmal type (often called Bot(tom)). In more advanced type sytems supporting parametric polymorphism exceptional values can be given the polymorphic type $\forall X.X$. This type can be instantiated to any other type. C++ supports polymorphic types as do dialects of Java.

An important characteristic of imperative languages is the ability to have side-effects. Side effects mean a call to a function can have several effects, **besides** the return value. An example of these side-effects are **void** methods in Java. A method can return a value of the type **void**. This is essentially a bottom value, indicating nothing. The method however can have the effect of printing a message to the screen.

4.5 Summary

This chapter showed how exceptions are often used in the class of imperative languages. We showed that exceptions are used for abstracting errors. We showed the basic syntactic elements and how they are used. Furthermore we showed how exceptions are combined with the type-systems. Finally, we showed how exceptions are possible in this class of languages due to side-effects. Exceptions can also be regarded as a side-effect of a function. Side effects therefore allow an easy addition of exceptions to a language.

Chapter 5

Exception handling in strict functional languages

In chapter 4 we have seen how exceptions are used in imperative programming languages. We have also seen how exceptions were designed. In this chapter a description is given of the design of exception handling in strict (non-lazy) functional programming language. We will see that these languages have other problems and solutions for exception handling.

5.1 Definition of strict functional languages

This chapter discusses strict functional languages. Strict functional languages are defined as language that use call-by-value semantics and don't use closures. Strict language use side-effect for IO. The use of side-effects allows an easy implementation of exceptions, for example using continuations.

This chapter uses the languages OcaML and Erlang for examples. Both are strict functional languages used in production quality software systems.

Describe how exceptions are handled in a strict functional language. Especially pay interest to side effects and type systems.

5.2 Examples of exception handling syntax and semantics

Exception handling in strict languages is still rather new. Since object support was added to caML, in 1996, exceptions are supported. Erlang

recently redesigned their exception handling system[CGN04]. The original version was incorporated into Erlang in 1996[]. Discussed here is the new design, the older design is discussed in [vdB03].

5.2.1 throw and catch

According to [CGN04], exceptions are raised in Erlang using the keyword **throw** and caught using **catch**. As indicated by [CGN04], the exception mechanism of Erlang has a number of problems. Exceptions are part range of return values of each function that raises an exception. A pre-defined tuple

```
{'EXIT', Reason}
```

is defined as an "exceptional" value.

Listing 5.1: Erlang's exception handling

```
case catch F(X) of
  {'Exit',Reason} -> handle(Reason)
end.
```

In OcaML exceptions are thrown using **raise**.

Listing 5.2: OcaMLs exception matching

```
try E with
| p1 -> exp1
| p2 -> exp2
exp2 = raise E
```

Catching exceptions in OcaML is done using patternmatching. See the example below (listing 5.3).

Listing 5.3: OcaMLs exception matching

```
#let name_of_binary_digit digit =
   try
     List.assoc digit [0, "zero"; 1, "one"]
   with Not_found ->
     "not a binary digit";;
```

5.3 Examples of usage

In general the use of exceptions in strict functional languages is similar to the use of exception handling in imperative languages. Although functional languages return exceptions as part of the return value of a function, the general techniques are the same.

5.3.1 Head of empty list

This example shows how to handle a head of empty list using OcaML's exceptions.

Listing 5.4: Head of empty list

```
exception EmptyList ;;
let head = function
  [] -> raise EmptyList
  | x::xs -> x ;;
```

Running head [] returns an exception: EmptyList.

5.3.2 Division by zero

To show OcaML doesnt return lists with exceptional values in it, we use a list of possible exceptional values:

Listing 5.5: Division by zero and lists

```
exception FaultyContent ;;
exception DivByZero ;;

let operator = function
 0 ->  raise DivByZero
 | n -> 10 / n ;;

let rec procesList ls =
 List.map operator  ls ;;

let rec take n s =
  match s with
    [] ->
      (match n with
        0 -> []
```

```
      | n -> raise FaultyContent)
  | (x::y) ->
      (match n with
        0 -> []
      | n when (n < 0) -> raise FaultyContent
      | _ ->
         x :: (take (n - 1) y)) ;;

let main =
 take 3 procesList [1;2;3;0]
```

This function raises the exception DivByZero. In the next chapter we will see that this code doesn't throw an exception in Haskell.

5.3.3 Adding two exceptions

To demonstrate that OcaML does make a deterministic choice between two exceptions we constructed the following demo program. Two function each return an exception and are combined using +.

Listing 5.6: Referential transparency and exception addition

```
exception OP of string;;

let operator1 x y =
 match y with
 0 -> raise (OP("op1 error"))
 | _ -> x/y;;

let operator2 s =
 match s with
 [] -> raise (OP("op2 error"))
 | (x::xs) -> x + List.length xs;;

main = operator2 [] + operator1 3 0
main = operator1 3 0 + operator2 []
```

This example returns another exception at each call of **main**. The first call returns: Exception: OP "op1 error", the second: Exception: OP "op2 error". Evaluation stops when an exception is raised, and control is transferred to the handler.

5.4 Type system details

By definition, error terms or exception make it harder to construct a typing algorithm. Since any function can return an exception, an exception is of **any** type! Therefore one of the most fundamental properties of typing algorithms is violated: every typeable term in the language has a unique type. Furthermore evaluation and progress theorems are violated. In a language with exceptions, a term does not always evaluate to a value normal form.

There are a number of options to fix these problems. Depending on the typing system used by the language and the evaluation strategy, be it eager or lazy. In this chapter we will only consider eager strategies and various typing systems.

One of the solutions to ignore the type of exceptions that are raised. Only the catch function checks if its arguments are of the type of exception the function expects.

Due to the evaluation order being fixed at runtime (caused by eager evaluation) we can determine what exception was raised first. For example this code (listing 5.7):

Listing 5.7: Deterministic exception catching

```
expr = raise E + (1/0)
exp2 =
  try expr with
  | E -> print "Exception"
  | DivisionByZero -> print "division by zero"
```

This example will always print "Exception".

5.5 Difference with imperative languages

The main difference between exceptions in strict functional languages and imperative languages is the use of exceptions as return values. In imperative languages, exceptions are used as controlflow elements. Upon throwing an exception, control is transferred to the handler. In functional languages, exceptions are often return values of functions, as shown in the examples using Erlang above.

5.6 Concluding remarks

We described the designs and implementations of exception mechanisms in strict functional languages. In the next chapter we investigate the problems caused by lazy evaluation strategies in combination with exception handling.

Chapter 6

Problems with lazy functional languages

In the previous chapters we discussed how exceptions are used and implemented in object oriented languages and strict functional languages. Before we discuss the current state of exception handling in lazy functional languages, we need a detailed description of the problems that arise when designing exception mechanisms for lazy functional languages. This discussion sheds some light on the design choices made by the designers of current exception mechanisms in Haskell.

There are a number of major problems with the use of exception handling in a lazy functional language. First, lazy evaluation does not allow us to know beforehand the order in which expressions are evaluated. This is due to the demand-driven evaluation. Therefore we cannot use the way an expression is evaluated to reason about exceptions. Exceptions are part of the return-value of an expression. Furthermore we cannot detect what exception happened first in an evaluation. We cannot reconstruct the "history" of evaluation without help of some auxiliary mechanism.

Second, we need to leave referential transparency unchanged. This is an important property of functional languages and is used for numerous transformations in order to improve efficiency.

Third, we need to find a balance between easy understandable semantics, a nice uncluttered syntax and implementation. The most important part of the implementation of exceptions in lazy functional languages is that the solution is actually usable by the average functional programmer.

6.1 Problems

a) Referential transparency

One of the core values of functional programming is referential transparency. Referential transparency can be defined like this: the relation between the input and output of a function is always the same, no matter what evaluation order or outside influences. Exceptions returned as values violate this definition, they can return values upon error that are not part of the normal input/output relation.

b) Lazy evaluation and discriminating exceptions

Due to the mechanism used in lazy functional languages to determine what value is needed at any point of the function graph, it is impossible to determine deterministically which exception occurred further down the function graph. It is non-determined what set of exceptions each implementation of a language with lazy evaluation semantics returns on the same situation at runtime. One of the examples of this problem is the +-problem, described in paragraph 6.2.

c) Maintaining compiler and runtime efficiency

Modern compilers for functional languages use a number of transformations at compile-time to generate more efficient code. These transformations are often based on the notion of referential transparency (described earlier in this document). Exception handling used carelessly breaks these transformations.

d) Clear semantics

Another important issue regarding exceptions in lazy functional languages are the semantics of the design. Clear semantics are important for a number of reasons. First of all it precisely describes the behavior of the exception mechanism. This way, the user always knows how a statement will evaluate.

6.2 Examples of problem cases

This section describes a number of cases which refer to conceptual exception-handling. These are all problematic cases where the problem described in

paragraph 6.1 occur. All problems arise from using exceptions in a lazy context.

a) The + problem, referential transparency

Listing 6.1: The + problem

```
getException ((1/0) + error "Urk"))
```

This example describes the problem of the selection of exceptions. Both expressions of the + statement throw an exception. How do we know which exception to choose? Is it **DivideByZero** or **UserError "Urk"**? We can regard this as a set of exceptions. Also we would like **a + b** to return the same value as **b + a**.

Thus this problem actually represents two problems: choosing an exception from a number of exceptions in an evaluation graph and maintaining referential transparency.

b) The propagation problem

Listing 6.2: Lazy evaluation problem

```
zipWith f [] [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f xs yx = error "Unequal lists"
```

In this case exceptions may "hide" in unevaluated expressions. Here **zipWith** may return a list with values and an exception at the end of the list. Furthermore it may also return a complete list whose elements contain an exception. Example taken from [JRH+99].

c) Keeping transformations valid

One of the core features of lazy functional languages is that, during development, one can transform an ineffective specification of a program to an effective one. These translations can also be done automatically by the optimizing compilers. An important issue of exception handling is to keep these transformations valid.

6.3 Repercussions for language design

If one wants to add exception handling in a semantically clear and runtime efficient way, one needs to address the problems described in the previous paragraphs. Any possible solution will have a deep impact on the language properties, design and performance. Exception handling requires changes in the runtime system (for example to allow handler frames to be installed on the stack), the compiler (certain transformations can no longer be performed) and the syntax.

This integration with all facets of language design makes exception handling in lazy functional languages a complex issue. The challenge is to find the right balance between performance, user friendliness and clear semantics. In the next chapter we describe the current state of exception handling in lazy functional languages, e.g. Clean and Haskell.

Chapter 7

Current implementations of exception handling in lazy functional languages

This chapter describes the currently available implementations of exception handling in lazy, pure functional languages. Two implementations were found during the research: imprecise exceptions in Haskell[JRH+99], [MJMR01], [Unk] and imprecise exceptions in Clean[vWP03]. As these implementations are both based on the same design, they are both described here in the same paragraph.

7.1 Haskell's implementation and design

In [JRH+99] Simon Peyton Jones e.a. present a solution for exception handling in the lazy functional language Haskell[wc]. The key concept of Haskell's exception handling is the imprecise part. Semantically Haskell's implementation does not make a deterministic choice determining what exception to catch. Multiple executions of the same code may report different exceptions.

The design described in [JRH+99] is implemented in Haskell in the non Haskell '98 Control.Exception module.

The core idea of handling exceptions in current implementations of lazy functional languages is to address them as values. Exceptions are regarded as nothing more than a return value of a function. This idea is nothing new. Already exceptions are used as indicated by the following listing in many functional languages.

Listing 7.1: Exceptions as values

```
data ExVal a = OK a | Bad Exception
f :: Int -> ExVal Int
f x = ...definition of f...
```

Disadvantages of this solution are numerous:

case analysis Every function needs to check if ExVal a is Ok or an exception. This problem can be solved by using a Monad structure to reduce explicit passing.

increased strictness It is very easy to introduce increased strictness due to checking a function argument for errors when it is actually *passed* instead of *used*.

For other disadvantages we refer to the paper by Simon Peyton Jones e.a. [JRH+99].

7.1.1 Haskell's solution

To handle the problems described in 6.1 and 6.2 in Haskell the following solution was presented in [JRH+99].

Solving the propagation problem

One of the problems the design of Haskell exceptions did not fully solve is the propagation problem. The only way to make sure a data-structure does not contain any exceptions is to fully evaluate the data-structure using **seq**, a built-in haskell function.

"Solving" the + problem

The main problem to solve is the problem of the choice of exceptions versus lazy evaluation. The designers of the Haskell exception mechanism made a very interesting choice here. At any node in the evaluation graph of a program, one can regard all exceptions thrown in subnodes as a collection. As one has no knowledge of the order in which the exceptions were thrown, it is impossible to choose the "first" exception deterministically.

Naturally, explicitly passing the union of all exceptions thrown in subnodes would be a disaster to the efficiency of the implementation. Furthermore it would reduce laziness again! The solution for the problem chosen by the designers was to make the choice non-deterministic. By using a technique

pioneered by Hughes and O'Donnell[HO89], the union is represented and a choice is made. The key issue here is that the choice is **non-deterministic**. To achieve this, the function **getException** in Haskell is placed in the IO monad. By placing it in the IO monad, **getException** can theoretically consult outside sources to determine the correct exception.

Reference implementation

The implementation in Haskell operates as follows. **getException** marks the stack before evaluating its arguments to head-normal-form. **raise** simply trims the stack to the top-most getException mark, and returns Bad Ex as result of getException. If the evaluation of the argument of getException does not encounter a **raise**, **getException** returns OK val. (Quote from [JRH+99])

7.1.2 Changes in implementation

Exceptions were implemented as an extension to Haskell '98. They were implemented in GHC, the Glasgow Haskell Compiler. The implementation can be found in the module **Control.Exception**. This module contains the functions described in the paper by [JRH+99]: catch (contains getException), raise (called throw here). The implementors extended the original design with 3 main additions. They also precoded a number of default exceptions like *PatternMatchFail String, NoMethodError String* and *ArrayException ArrayException*.

1. **dynamics** Control.Exception extended the idea of exceptions with Dynamics, see [CH02]. Dynamics are a relatively new concept in functional languages with strict typing. A *dynamic*, allows of sending, storing and reloading data and even functions in a type safe way. Dynamics support runtime typesafe casting and construction of terms.

   Two new functions were added: **throwDyn, catchDyn** and a new exception: **DynException Dynamic**. Dynamics are used to enable user defined exceptions. The user defined exception can be anything that can be wrapped into a language.

Listing 7.2: Exceptions and Dynamics

```
throwDyn :: Typeable exception => exception -> b
catchDyn :: Typeable exception => IO a -> (
    exception -> IO a) -> IO a
```

```
--example
data DBException = DBError String | ZeroValues |
    ConnectionError Int String

getData :: String -> Database -> Result
getData s db = catchDyn (getResults s db)
    ( \dberror ->
    case dberror of
        DBError String -> {-handle general error
            -}
        ZeroValues  -> {-handle no data found-}
        ConnectionError -> {-handle db connection
            faillure-}
        _           -> {-rethrow/finish-}
    )
```

Dynamics were added to make the Exception datatype extensible. Using Dynamics, exceptions can be anything that's in the typeclass Typeable.**catchDyn** catches dynamic exceptions of the required type. All other exceptions are rethrown.

2. **xxxJust functions** Control.Exception contains a number of xxxJust functions. Examples are **tryJust** and **catchJust**. The signatures of these functions are described in listing 7.3.

Listing 7.3: xxxJust functions in Haskell exception handling

```
catchJust :: (Exception -> Maybe b) -> IO a -> (b
    -> IO a) -> IO a
handleJust :: (Exception -> Maybe b) -> ( b -> IO
    a) -> IO a -> IO a
tryJust :: (Exception -> Maybe b) ->  IO a -> IO (
    Either b a )
```

As indicated by the signatures of the functions, each function has a exception selector function, that matches against certain types of exceptions. Upon a match, depending on the function, the handler is called or something else is done. If the exception does *not* match, the exception is rethrown. There are a number of predefined exception selection predicates. These predicates are the exception selector functions.

Listing 7.4: Exception predicates

```
ioErrors :: Exception -> Maybe IOError
arithExceptions :: Exception -> Maybe
    ArithException
errorCalls :: Exception -> Maybe String
dynExceptions :: Exception -> Maybe Dynamic
assertions :: Exception -> Maybe String
asyncExceptions :: Exception -> Maybe
    AsyncException
userErrors :: Exception -> Maybe String
```

The Just functions can be used to selectively handle exceptions. For example when handling errors of a database connection, we are only interested in connection errors of the database and we do not want to handle exceptions like division by zero at that point.

3. **finally** The function **finally**[Unk] is modelled after the finally keyword found for example in the language Java. The following code snippet shows the syntax of this keyword in Java (listing 7.5).

Listing 7.5: Exception predicates

```
try {
    db = env.openDb();
    dbQueryResult = db.getResult(SqlQuery);
    //proces results
} catch ( SqlException ex ) {
    //handle the exception in some way
    ex.printStackTrace();
} finally {
    //even when something else fails.. close the
        db!
    db.close();
}
```

This example displays how finally can be used to clean up the environment after a computation has finished. Even in the event of errors in this computation, the finally block will be executed.

In Haskell, the function **finally** is defined as follows (listing 7.6):

Listing 7.6: Haskell's finally definition

```
-- finally, where IO a is the computation to run
   first
-- IO b is the computation to run afterwards, even
     in the event of faillure in IO a
finally :: IO a -> IO b -> IO a
```

As described by the comments in the code snippet, **finally** runs a computation, say a, until it either finishes or throws an exception. Afterwards another computation is ran, say b. The Haskell version of the java snippet (listing 7.5) would be something like this:

Listing 7.7: Haskell's finally example

```
--example dummy function that executes a query on
   a database
getDbQuery :: String -> db -> String
getDbQuery query database
    | query == ""    = throw DatabaseError "An␣
       empty␣SQL␣string␣was␣passed"
    | otherwise = --do something to create a
       result

Start = finally (getDbQuery query db) (dbClose db)
```

7.2 Clean's implementation

Clean's implementation of imprecise exceptions was demand driven. The implementation of a functional operating system shell[vWP03] needed a mechanism for handling errors in a non-terminating way. Due to the ad-hoc nature of the implementation it is still rather primitive. Clean equivalents of many of the utility functions of *Control.Exception* are missing.

The implementation is based on the paper by Simon Payton Jones e.a. [JRH+99], and follows the stack unwinding strategy. Clean uses the ABC-machine for its runtime execution. For details we refer to [PE93]. Catch is implemented as follows: a pointer is set to a special catch-node on the a-stack. The expression passed as a parameter to catch is evaluated to head-normal-form. After the expression is evaluated, the pointer is removed. This special catch-node contains the value of the b/c-stack pointer, the profile-stack pointer and a third pointer to the exception-handler.

On a raise, the a-stack is transferred, overwriting each cycle-in-spine-nodes, replacing them with the exception. When this stack transferal reaches a pointer to the catch-node, this catch block is evaluated. If the a-stack runs empty, a runtime message is printed showing the exception thrown. Evaluation of the catch block is started by replacing the value of the b/c-stack and profile-stack pointers by the values stored in the catch-node. Next the exception-handler found at the catch-node is evaluated using the dynamic argument of raise. After evaluation the exception handler returns to the catch following the address placed on the c-stack.

The implementation choice of overwriting cycle-in-spine-nodes helps to keep the implementation of exceptions deterministic. Each evaluation of a path gives rise to the same exception.

7.3 Disadvantages of the current exception handling mechanisms

The exception mechanisms of both Clean and Haskell suffer from a number of disadvantages. The designs function well in small examples or in typical functional code. Code where the use of **Maybe** can be augmented with exceptions.

1. The most important problem of these implementations is the requirement of **Catch** to have access to a world object or IO Monad to make it's non-deterministic choice. This has the disadvantage of pulling pure functional code that throws exceptions into the IO Monad with **evaluate**.

2. Furthermore, if you want to layer your software design and use different types of exceptions for each layer, you need to wrap each exception of a lower layer to an exception of the current layer. To do this, you need to use **Catch**, and thus you need to pass the IO monad all the way into your application. For example of this problem see listing 7.8 below. This code snippet shows that catch error needs the IO monad to catch and rethrow the exception.

Listing 7.8: Exception wrapping in Haskell

```
data  BusinessException =
      BError String
      deriving (Show, Typeable)
```

```
data DatabaseException =
      DBError String | DBUndefined
      deriving (Typeable, Show)

catchError :: IO a -> (DatabaseException -> IO a)
   -> IO a
catchError action handler = catchDyn action
      (\dberror ->
      case dberror of
        DBError e  -> throwDyn (BError "Business
           retrieval failed")
        _                  -> handler dberror
     )
```

3. The third problem is that due to the dynamic type system in Haskell the compiler doesnt notice type errors. As long as the argument to **throwDyn** is **Typeable**, the compiler will throw an exception. As an example view the code below (listing 7.9).

Listing 7.9: Problems throwing dynamics

```
--Define an exception to use in throwDyn
data MyException = MyError String

--the following 2 throw's do not throw the same
    exception

throwDyn MyError "help"

throwDyn (MyError "help")

--the compiler will not give a warning or error
    message as MyError "help" is also Typeable
```

4. Another problem is that uncaught exceptions, be they dynamic or otherwise typed, are just printed in the running console. One would like to see a stacktrace or any other diagnostic message. However the current exception system does not allow for a) storing contexts and b) easy wrapping of exceptions.

The next chapter elaborates on the question why we need exceptions and clean exception handling.

Chapter 8

Why do we need exceptions

In the last few chapters we have seen that exceptions and lazy functional languages don't match. The question is, why would we want to go to the trouble of inefficient code, less clear semantics, just to add a mechanism to handle errors? The introduction of exceptions to imperative, or more specific object oriented languages was driven by the need to create re-usable and modular code.

If lazy functional languages want to be used to create large scale professional software, they need to support an error abstraction mechanism. One of the main benefits of functional languages is compositionally (see Wadler [Wad92] or Hughes [Hug89]). If we can combine this with a good error handling mechanism this can lead to a language better suited for large information processing projects. Large projects like corporate information systems spanning multiple machines and departments need separation in modules. These modules consist of a clearly defined interface and clearly defined possible exceptions. This way application programmers aren't bothered by implementation details. An example is the use of a dataprovider module returning "Student not found" exceptions instead of "Empty record in database" or worse "Error -40001".

Besides error abstraction and enabling larger software systems, exceptions can also be used for non-terminating error reporting. For example, one of the problems the designers of the functional operating system shell ran into is the reporting and handling of errors without terminating the shell and thus the operating system. Exception handlers can be used as a last resort to display the error message to the user. Furthermore exceptions provide a clean way of raising and reporting errors. Other techniques like special return values or goto constructions do not force a single way of error

handling upon the programmer.

Another advantage of exceptions is a distinction between errors and wanted return values. For example, a commonly used C function is atoi, this function converts a char array to an integer value. If an error occurs during the conversion, the function returns 0. It is hard for a programmer to determine if the converted value was the actual value or an error during conversion.

Another more advanced feature of exception handling is the so-called "undo" support. This feature is still in a research phase. The idea is that upon the raising of an error, the exception handler tries to restore a failed pre-condition that caused the exception to be risen in the first place. This feature requires a clear definition of pre-conditions.

8.1 Next

This chapter showed that exceptions and their handling are paramount for the definition of clear programmers interfaces. These interfaces are needed for the development of large, enterprise scale, information systems. We also described a number of other features provided by exceptions. In the next chapter we present an alternative to Haskells exception handling for the lazy functional language Clean.

Chapter 9

Proposal: informal semantics and design

Our investigation of exception handling in other language classes showed us that exceptions are often integrated in the type system. In Java for example a form of type based pattern-matching is used to discriminate thrown exceptions. We also noticed many languages are able to use exceptions due to side-effects. Functions or methods can return exceptional values without obeying their type. The availability of side-effects allows for easy implementation of exception handling: stack unwinding as an implementation mechanism can be used.

As we have shown in previous chapters, none of these options are possible in a lazy, pure functional language. Therefore we investigated other ways of separating return types of the actual type of the function. In order to return exceptions from a function call we need a mechanism to add the exception type to the return values without disturbing the type system or evaluation mechanism.

To achieve this we introduce a "co-domain"[1] to store non-normal return values. We must take special care of the typing of the co-domain, as careless typing will disturb the typing algorithm or, more specific: the uniqueness of types theorem[Pie02]. This theorem states that: "Each term t has at most one type. That is, if t is typeable, then its type is unique". Exceptions can be of any type, usually strings are used to carry the message of the exceptions. We need to define our types in such a way that all these exception types are possible. Other types are possible, for example to support a "repair" or

[1] co-domain in this context means a disjoint domain of typed values, disjoint from the regular value domain

undo mechanism. A recent addition to the Clean language comes into focus here: dynamics[vWP03] e.a.. Using dynamics we can incorporate a separate dynamic typing system besides the usual static typing system.

We can use a custom type to wrap normal types and exceptional returns, a so called SafeType. This way we introduce our errordomain without explicitly needing dynamics. In order to support user defined exceptions, we do need dynamics.

9.1 Informal design

This paragraph explains the design of the exception mechanism in an informal way. We will describe the general method of handling errors and sketch the design. The design is centered around the thought of 2 domains; one for normal operation and another one for erroneous situations. The central point of this design is the mapping from one domain to the other.

At first we describe the two domains. Key to the mechanism is every function returns, conceptually, a tuple of two domains. The first domain is the normal, statically typed function domain. This contains the value of the evaluation, if no errors occurred during evaluation. The second domain represents the error domain. It consists of a type that has the property that it's appendable and has a zero information element. Appendable in this context means there is some kind of add function that joins two elements of this type into one element. Examples are string or list and dynamic. Our safeType contains the indicator of the error. We can regard this as the error administration. If the appendable type is a dynamic, it can contain various forms of error administration. For instance: exception handling code, simple string message forms or debugging code. In case of an error we change to the exception or error domain and nullify the normal domain by using the $\perp$ function, indicating the returned value has no definable type.

Besides these two domains we need a mapping from the normal functional, strictly typed domain to the error domain. These mappings need to be clear to the application programmer and should adhere to common syntactical forms like throw and catch. In this context, throw represents a transition from the normal domain to the error domain. In contrast, catch represent the opposite: return from the error domain to the normal domain while handling the error.

To determine in which domain we are operating we need some kind of flag to indicate error or normal domain. This flag is implemented as a simple boolean variable at this time. Please refer to the next paragraph for a more

detailed description of this flag.

The real power of this system is the availability of a dynamic error domain. Due to its dynamic typing, this system can contain almost everything. We can use it to write debug strings to a text file or try to restore a precondition of a function. The possibilities are endless!

9.2 Defining the semantics

In this section we will describe the semantics of our proposal. We use the structure of the lambda-calculus[Bar84]. Its primary structures will be leading: application and abstraction. At first a minimal language is defined. Using this minimal language mappings are defined from the minimal language without exception handling to an augmented version of this minimal language including exception handling. The primary aim of this paragraph is not to provide a complete and correct semantics, it rather is to make it plausible a complete and correct semantics exists. A full operational semantics are subject for further research.

9.2.1 Definition of a minimal language

$$
\begin{array}{ll}
e := x & \text{//variable} \\
\quad |k & \text{//constant} \\
\quad |e_1 e_2 & \text{//application} \\
\quad |\lambda x.e & \text{//abstraction} \\
\quad |C\ e_i \ldots e_n & \text{//constructors} \\
\quad |case\ c\ of\ \{\ldots p_i \rightarrow e_i; \ldots\} & \text{//case} \\
\quad |e_1 + e_2 & \text{//primitives} \\
\quad |e_1 - e_2 & \text{//primitives} \\
\quad |e_1 / e_2 & \text{//primitives} \\
 & \\
P := C\ x_i \ldots x_n \mid \_ & \text{//pattern}
\end{array}
$$

This minimal language supports a call by need evaluation method and is modeled after Clean. To incorporate exception handling we need to add a number of features to this language. First we need simple type support. For demonstration purposes we will only include the following basic types: Boolean, Int. Furthermore we support algebraic types. Second we need our

language to support some output type: string. For sake of the discussion semantics and type evaluation rules are left out. As a remark we point out that the +, -, / operators are strict, i.e. they evaluate their arguments before evaluating themselves.

In this language we use **undef** to indicate the element containing the least information of the chosen domain. I.e. undef is equal to bottom.

$$type := Bool|Int|string \tag{9.1}$$

Furthermore it is allowed to name abstractions, i.e. use function names for abstractions. For example:

```
addOne x = \y.y+1
```

9.2.2 Example code of the mini language

In this paragraph we will give some examples of program code in the language defined above. We discuss the behavior of the program.

Listing 9.1: Example 1: calculations

```
calculation x y = \x.y.(x+y)/(x-y)

lookAtResult x = \x.
    case x of
    0 -> true
    1 -> false
    _ -> false

lookAtResult (calculation 1 2)
```

This example code will crash when the inputs are equal to each-other, resulting in a division by zero. If input x and y of lookAtResult are not equal, this function will return a calculated value.

9.3 Adding exceptions

To add exceptions to this mini-language we introduce a number of additions to the language. At first each variable is transformed to a new type: the ST type.

Listing 9.2: The ST type

```
:: ST a = ST Status a ExcType
:: Status = Correct | Error
:: ExcType :== string
emptyExc :: ExcType
emptyExc = ""
```

For the sake of the mini-language, exceptions are just strings.

In order to lift basic values and functions to the SafeType domain we introduce the function lift:

Listing 9.3: The lift function

```
lift :: a -> (ST a)
lift a = ST Correct a emptyExc
```

Besides this addition we also need a way to transfer to and from the error domain. In order to achieve this we add a throw and a catch function.

Listing 9.4: raise

```
raise :: ExcType -> ST a
raise exc = ST Error undef exc
```

Using raise we can throw an exception: transfer a value to the error domain. This is done by setting the domain indicator to Error.

To transfer from the error domain to the normal value domain we need a function to handle an error and reset the domain indicator.

Listing 9.5: catch

```
catch :: (ST (ExcType -> (ST a))) (ST a) -> (ST a)
catch handler (ST x y exc) =
    case x of
        Correct -> (ST Correct y exc)
        Error -> AppToExc (handler exc)

AppToExc :: (ST (ExcType -> (ST a))) ExcType -> (ST a)
AppToExc (ST x handler ex) exc=
    case x of
        Error -> raise ex
        Correct -> handler exc
```

This design uses **AppToExc** to prevent the handler or catch from throwing errors that are not in the exception mechanism. Catch detects the value of

the evaluated input function, and thus forces evaluation of the input to head normal form. **AppToExc** checks if the handler itself threw an exception. If an Error status is detected, The handler is evaluated, resulting in a reset of the Error domain indicator and an empty exception.

Now that we have the constructions to raise and catch a predefined exceptions we investigate the propagation of exceptions. At any function call, if there is no catch in the function, we need to stay in the error domain and propagate the exception(s) to the next call. Function application is one of the basic operations of the lambda calculus and is explained in more detail in the following paragraph.

Being able to raise custom exceptions implies that at any point in a function call graph, multiple exceptions exist at a node in the call graph. In order to propagate all exceptions, we need a combinator to join these exceptions to a set. We introduce the function UN for our example string exception scheme:

Listing 9.6: UN function, for uniting exceptions

```
(UN) :: ExcType ExcType -> ExcType
(UN) a b = a +++ " FOLLOWED BY " +++ b
```

See paragraph 9.4.2 for a more detailed description of the UN function. Even though our mini language does not support infix operators, a conversion from infix operators to Polish notation is trivial and therefore not discussed here.

9.4 Mapping the mini language to the mini language with exceptions

In the following paragraphs we will define mappings from the minimal language to the minimal language augmented with exceptions. Each paragraph defines one of the basic constructions of the language. For brevity, we leave out the λ abstraction upon function definition.

9.4.1 Variables

Variables are wrapped into the SafeType domain.

Listing 9.7: variables

```
x := ST Error x emptyExc
//or:
lift x
```

Constants are handled in the same way as variables. Constants do not contain exceptions, and are thus always in the "Correct" domain.

9.4.2 Application

Application is one of the basic operations in the lambda-calculus:

$$F.A$$

or

$$FA$$

this denotes the data F considered as algorithm applied to the data A considered as input.

In general upon application of a function to its arguments, there can be two situations: either we are in the error domain or we are in the correct, value domain. If we are in the error domain, indicated by one of the arguments to the function, we need to propagate the exception(s).

If we want to apply our exception design to this operation, we can use case distinction. Apply consists of two degrees of freedom: the algorithm or function F and the input A. Both can be in the error domain. Therefore we need to inspect both of them for errors, using our predefined ErrorStatus. It's important to notice when this inspection of ErrorStatus is done. Following the call-by-need evaluation semantics, inspection of ErrorStatus should be done when the **value** of an expression is required.

Listing 9.8: non exception application

```
apply :: (a -> b) a -> b
apply f a = fa
```

Listing 9.9: application

```
apply :: (ST ((ST a) -> (ST b))) (ST a) -> (ST b)
apply f a =
    case f of
    ST Error _ ex -> raise ex
    ST Correct f _ -> f a
```

The function argument of apply is inspected for erroneous values. If an error is found a new entry is placed in the error domain, consisting of the raising of a exception.

Notice that functions should take their own responsibility of throwing errors. For example in listing 9.11 the function application (f a) can still cause errors that are not caught by the exception mechanism. We cannot detect failure of these functions as Turing has shown we cannot detect non-termination. To indicate functions themselves can throw exceptions we made the type of the function: $(ST\,((ST\,a) \rightarrow (ST\,b)))$, indicating that we expect a function from SafeType a to SafeType b, that is able to throw exceptions, hence the ST wrapper.

Therefore, we do not expect the arguments of the function on apply, the function itself handles the exceptions and their propagation.

Another point of interest is the UN function, UN stands for unification. This function combines two exceptions. It is up to the program writer to define the UN function as the program writer is free to use whatever (s)he likes as exception. UN is used when a function has exceptions in its arguments and wants to raise an exception itself. In the case of our string example of excType we get the UN function defined in listing 9.6

We realize our mini language does not support either concatenation or infix operators. We assume the reader finds it plausible that adding these features to the mini language does not disturb the exception mechanism.

9.4.3 Abstraction

Abstraction is said to bind the free variable x in M, where M is a lambda term. E.g. we say that $\lambda x.yx$ has x as bound and y as free variable. Substitution [x := N] is only performed in the free occurrences of

$$x : yx(\lambda x.x)[x := N] \equiv yN(\lambda x.x).$$

This example is taken from [Bar88].

In the context of the proposed exception handling system this translates to implicit exception propagation. Essentially, lambda abstraction is used for defining functions.

Listing 9.10: non exception abstraction

```
apply (\x.x+1) 2
```

Where \ is λ in listing 9.10.

The process of abstraction itself does not introduce new exceptions. In the mini-language the operation of abstraction should propagate exceptions that occurred earlier. This however is automatic, by design. If an error occurred, the system is in the the error domain.

9.4.4 case

Case evaluates the patterns in a top-down fashion. It evaluates the argument a of case and compares this top-down to the patterns p. Evaluation of argument a may result in an exception. In this case the pattern should cover this situation. Furthermore each pattern p may contain an exception. These should also be propagated.

Listing 9.11: case, patterns cover error situation

```
safeCase checkVal of
    ST Error _ exc  = raise exc // propagate exception
    ST _ val _      = ...
```

We cannot implement the case in our mini language without using lists to check all possible patterns p for exceptions. We cannot use the normal case as case can have any number of arguments as patterns. We leave safeCase unimplemented in our mini-language and show the an implementation of safeCase in our prototype implementation in chapter 9.1.

9.4.5 primitives

This paragraph describes the changes needed to support exception handling for the basic arithmetic operators. These operators are evaluated in a strict matter.

operator -

The minus operator checks both arguments for the presence of exceptions. In case both arguments contain exceptions, the exceptions are united using the pre-defined UN function.

Listing 9.12: non exception - operator

```
- :: Int Int -> Int - x y = x - y
```

Listing 9.13: - operator

```
safe- :: (ST Int) (ST Int) -> (ST Int)
safe- x y =
    case x of
    ST Correct a _ ->
        case y of
        ST Correct b  _ -> lift (a - b)
```

```
        ST Error _ exc -> raise exc
    ST Error _ d ->
        case  y of
        ST Correct b _ -> raise d
        ST Error c e -> ST Error undef (d UN e)
```

In this listing (9.13)"-" indicates a regular - operation on integers. As we can see in the last case, where both arguments have an exception, we use the UN function to unite the two exceptions.

operator +

The plus operator is defined in the same matter as the minus operator.

Listing 9.14: non exception + operator

```
+ :: Int Int -> Int + x y = x - y
```

Listing 9.15: + operator

```
safe+ :: (ST Int) (ST Int) -> (ST Int)
safe+ x y =
    case x of
    ST Correct a _ ->
        case y of
        ST Correct b  d -> ST Correct (a + b) d
        ST Error c d -> ST Error undef (d UN 2 )
    ST Error _ d ->
        case  y of
        ST Correct b e -> ST Error undef d
        ST Error c e -> ST Error undef (d UN e)
```

operator /

The divide operator is an example of a function that generates exceptions: division by zero can occur when the second argument is zero. This introduces a second check. See listing 9.17.

Listing 9.16: non exception / operator

```
/ :: Int Int -> Int / x y = x / y
```

Listing 9.17: / operator

```
safe/ :: (ST Int) (ST Int) -> (ST Int)
safe/ x y =
    case x of
    ST Correct a _ ->
        case y of
        ST Correct b  d ->
            case b of
            0 -> raise "division_by_zero"
            _ -> lift (a / b)
        ST Error c d -> ST Error undef (d UN 2 )
    ST Error _ d ->
        case  y of
        ST Correct b _ -> raise d
        ST Error c e -> ST Error undef (d UN e)
```

As shown in the previous definitions, there is a recurring pattern in the application of the exception mechanism. At each call the state indicator, the Correct, Error, Unevaluated triplet, is inspected at the time the value of the variable contained in the wrapper ST is needed. This is important. If the value of the state indicator is inspected at any other time, laziness is reduced. In case more then one exception is found, the error domain is filled using the UN combinator, combining errors into one exception.

In the next chapter we introduce a prototype implementation written in Clean. We show how to implement the design described in this chapter, with exception of the runtime support.

Chapter 10

Proposal: Prototype implementation in Clean

In this chapter we investigate the prototype implementation[1] of the exception scheme and design described in chapter 9.

10.1 Implementation steps

In order to implement the design of the previous chapter, we need to make a number of adjustments to Clean. At first all types need to be transformed to the SafeType wrapped type, according to the scheme given in chapter 9. Furthermore all built-in primitive functions (of StdEnv) must be changed to support handling SafeTypes and their exceptions. Instead of implementing this in the compiler and runtime system and existing libraries, we opted for a prototyping approach. Using existing Clean functions, we created an exception mechanism on top of Clean. Even though it misses syntactic sugar, the prototype shows the possibilities of the design. The complete source of the prototype is supplied as an appendix to this thesis (Appendix A)[2].

10.2 Providing an exception scheme

In order to provide an implementation that is actually usable for the application programmer, we need to define exceptions in a concise manner.

[1] downloadable from `http://hypernation.net/clean/` or see Appendix A

[2] In order to improve readability all uniqueness attributes and other type attributes have been removed

To provide flexibility for the application programmer, he/she can define a collection of exceptions for each clean module. Clean already knows two types of files: .dcl for definition of functions and .icl for implementation of these functions. We want to add another type of file containing the exceptions used in the module and the union function to join them. We also need a function defining the value of an empty exception.

Any function that raises it's own exceptions must declare those in this file. In our prototype implementation we use the notation

```
functionname = doSfunctioname exc+
```

to declare these exceptions.

The aim of this file is to provide a way to make error handling type-safe. Using the definition of the exception, that is to be placed in the dynamic, and the Union function used to join exceptions in the ErrorDomain, we can determine if error handling is type safe.

An example of an exception scheme for strings is:

Listing 10.1: Example of an implementation of a exception scheme

```
:: ExcType :== String

(UN) :: ExcType ExcType -> ExcType
(UN) a b = a +++ " FOLLOWED BY " +++ b

emptyExc :: ExcType
emptyExc = ""

sDiv :: (ST ((ST Int) -> ((ST Int) -> ST Int)))
sDiv  = doSDiv "exception in sDiv: divide by zero"
```

the function **doSDiv** in listing 10.1 defines the exception, here a string, that the function div may raise. In a similar manner, the application programmer can define exceptions for other functions.

A programmer can use **any** type as exception, excType that has the following properties:

1. Has an empty instance, for example "" for strings
2. Is composable, two instances can be joined to one, for example dynamics

We refer to appendix A for a complete implementation in Clean.

The availability of a user definable excType and UN function are important. It enables our design to use exceptions for example as a repair mechanism. Furthermore exceptions can be customized per module.

10.2.1 Implementation in a compiler

In order to provide a usable exceptions mechanism for programmers, the process of lifting functions and variables to the safeType system should be done automatically. We imagine a compiler is able to make this transfer in either a source-to-source translation or in an advanced form of generic programming.

The application programmer should provide the exception scheme as explained in paragraph 10.2. The combining of the converted functions and the UN and exception definitions provided by the programmer lead to a working exception mechanism.

10.3 Example: providing a safe Case function

In chapter 9, we showed how we designed our exception mechanism. We noticed we could not fully implement a case function that checked each pattern for exceptions. This was due to the deliberately leaving out of lists in our mini language. In our prototype however, we have the full feature set of Clean to our disposal.

We implemented the following safeCase:

Listing 10.2: safeCase

```
doSCaseOf :: ExcType (ST a) [(ST a, ST b)] (Result b)
   -> ST b
doSCaseOf caseOfExc checkVal listOfCases defaultResult
= case checkVal of
    ST Error _ exc      = raise exc // propagate
        exception
    ST Correct val _    = matchCase val listOfCases
        where
        matchCase _ [] = case defaultResult of //
            check whether there is a defaultResult
            Default x   = x
            NoDefault   = raise caseOfExc // otherwise
                raise caseOfExc
```

```
matchCase matchVal [(a, result) :
   restListOfCases] = case a of
    ST Error _ exc  = raise exc
    ST Correct a _
        | val == a  = result
        // next element of list of cases
        | otherwise = matchCase matchVal
            restListOfCases
```

Here the arguments of the **doSCaseOf** are the element to evaluate, and a list of tuples of patterns and functions. Each pattern in a tuple element is checked for exceptions.

10.4 Examples of the prototype

This paragraph show some of the more appealing uses of the exception mechanism described in this thesis. The examples are given in Clean itself, using the prototype implementation. This implementation lacks runtime support and thus uses an indicator for the choice of domain. These examples show how this design can solve problems described in Chapter 6. Our reference implementation doesn't support curried apply. Therefore we created a number of functions to apply a function to a fixed number of arguments. Appendix A shows this example implementation and the functions defined therein.

To support the examples we defined the following catch and raise functions:

Listing 10.3: Catch and raise in Clean

```
catch :: (ExcType -> Bool) (ST (ExcType -> (ST a))) (
   ST a) -> (ST a)
catch _ _ arg=:(ST Correct _ _)              = arg
catch catcheable handler (ST Error _ exc)   =
 if (catcheable exc) (AppToExc handler exc) (raise exc
    )

// raise covers also unique variant
raise :: ExcType -> (ST a)
raise exc = ST Error undef exc
```

At first we show how our implementation handles the examples given in Simon Peyton Jones his paper[JRH+99].

10.4.1 +

The + problem in [JRH+99] is stated like this:

Listing 10.4: + problem of Simon Peyton Jones

```
getException :: a -> ExVal a

getException ((1/0) + (error "Urk"))
//where error s raises an exception
```

The problem is that an implementation of exception handling can either raise division by zero as an exception or "Urk". Where the Haskell solution combines both exceptions in a set, a clever semantics trick allows them to only show one exception, depending on what the oracle in the IO monad says. This doesn't really solve the problem, even though it's cleverly avoided.

Our implementation nicely returns the set of exceptions as combined with the UN combinator.

Listing 10.5: + problem of Simon Peyton Jones

```
displayExpr ((raise "urk") + (sOne/sZero))
//result is: "urk FOLLOWED BY exception in sDiv:
    divide by zero"

displayExpr ((sOne/sZero) + (raise "urk"))
//result is: "exception in sDiv: divide by zero
   FOLLOWED BY urk"
```

Once again we refer to appendix A for a full source of our implementation.

10.4.2 ZipWith

Another problem in [JRH+99] has to do with preserving laziness and automatic propagation of exceptions. Quoted from [JRH+99]: "A call to zipWith may return an exception value – for example, zipWith (+) [] [1]. It may also return a list with an exception value at the end– for example, zipWith (+) [1] [1,2]. Finally, it may deliver a list whose spine is fully defined, but some of whose elements are exceptional values– for example zipWith (/) [1,2] [1,0]"

We will show our implementation of zipWith and the application of the function to all situations mentioned in the quote.

Listing 10.6: zipWith problem of Simon Peyton Jones

```
sZipWith :: .(ST ((ST ((ST a) -> (ST b) -> (ST c))) ->
    (ST (List (ST a))) -> (ST (List (ST b))) -> ST (
   List (ST c))))
sZipWith = lift zipWith'
    where
    zipWith' :: (ST ((ST a) -> (ST b) -> (ST c))) (ST
       (List (ST a))) (ST (List (ST b))) -> ST (List (
       ST c))
    zipWith' sFunc sLista sListb = sCaseOf sLista
        [(sNil, checkListbEmpty)]
        (Result computeRest)
        NoDefault
        where
        checkListbEmpty = sCaseOf sListb
            [(sNil, sNil)]
            (Result (raise "Unequal list sizes"))
            NoDefault
        computeRest = sCaseOf sListb
            [(sNil, checkListaEmpty)]
            (Result (App2 sCons func (App3 sZipWith
               sFunc tla tlb)))
            NoDefault
            where
            func = App2 sFunc hda hdb
            hda = App1 sHd sLista
            hdb = App1 sHd sListb
            tla = App1 sTl sLista
            tlb = App1 sTl sListb
        checkListaEmpty = sCaseOf sLista
            [(sNil, sNil)]
            (Result (raise "Unequal list sizes"))
            NoDefault


//the examples SPJ gives:

//list definitions:
singleOne = listToSaveList [sOne]
emptyList = sNil
oneTwo = listToSaveList [sOne, sTwo]
```

```
oneZero = listToSaveList [sOne, sZero]

//application:
sZipWith (+) emptyList singleOne
sZipWith (+) singleOne oneTwo
sZipWith (/) oneTwo oneZero

//results:
["Unequal list sizes"],
[2,"Unequal list sizes"],
[1,"exception in sDiv: divide by zero"]
```

10.4.3 Wrapping

Wrapping is the conversion of one kind of exception to another. In our design exceptions, are encoded in the excType. Wrapping an exception is simply a matter of determining if an exception is present in the SafeType, and if it is, replace it by another exception. This replacement exception "wraps" the earlier exception. If a dynamic is used as excType, we can even put the handler code in the dynamic.

As an example we give a Clean version of the C# example of chapter 4, listing 4.2.

Listing 10.7: Exception wrapping using the new design

```
DivideByZeroException = "Division by zero"
MethodErrorException =
"Error in myclients method call"

doMethod :: a -> (ST ((ST Int) -> ST Int))
doMethod divByZeroExc
= lift method
    where
    method :: (ST .Int) -> ST Int
    method x = sCaseOf x
        [(sZero, raise DivideByZeroException)]
        (Result (App2 sDiv (lift 100) x))
        NoDefault

sMethod :: (ST ((ST Int) -> ST Int))
sMethod = doMethod DivideByZeroException
```

```
//define the function calling method, wrapping the
    exception to another one
doMyclient :: String -> (ST ((ST a) -> (ST a)))
doMyclient methodError = lift myclient
    where
    myclient :: u:(ST a) -> v:(ST a), [u <= v]
    myclient f = catch always (\x -> raise methodError
        ) f

sClient :: (ST ((ST a) -> (ST a)))
sClient = doMyclient MethodErrorException

Start = (displayExpr (App1 sClient (App1 sMethod (lift
    0))))
```

Executing this program gives us:

```
"Error in myclients method call"
```

showing the wrapping of the division by zero exception to the exception raised by the client.

Please note the use of the doFunctionname nomenclature to indicate functions raising exceptions. The function displayExpr simply prints a string value to the screen. Other interesting parts of this example is the use of App1, the apply function. Due to the lack of currying support in our prototype, this function is used to lift function application to the SafeType domain.

Notice we do not need any reference to a world object or other external sources to inspect for the exception, or wrap it.

10.4.4 Collection of exceptions

One of the main problems of the solution of Simon Peyton Jones e.a. [JRH+99] is the disability to determine what exception occurred. Their solution was to make an undetermined choice of a set of exceptions represented by placing the catch function in the IO Monad. The solution described in this chapter allows us to collect all exceptions, messages of exception or handlers per exception in our error domain. This implies we do not have to make a choice. We essentially create a set of all exceptions.

As an example we show the application of a list of functions to a list of arguments, using the function sum. Each application could throw an exception, resulting in a collection of exceptions or a single value if there is no exception.

Listing 10.8: Exception accumulation

```
sum = App1 sSum myList

sum' = App1 sSum myCorrectList

myList =
  listToSaveList [sZero, sOne/sZero,sOne/sZero,sOne/
     sZero,sOne/sZero, sOne, sTen,
  sOneHundred]

myCorrectList = listToSaveList [sZero, sOne, sTen]

//handler, determines the size of the exception set
handler :: ExcType -> *(ST Int)
handler x = lift ( length x )

//catch
catch always (lift handler) sum
catch always (lift handler) sum'

//result: 4 11
```

This example uses the alternative administration of example 10.10. This uses a list to store the exceptions.

10.5 Alternative administrations

The use of a dynamic as ExcType allows for more flexibility for the programmer. Instead of having just one exception handler somewhere in the code that handles a specific exception, we can store pieces of program code in our error domain dynamic. These pieces allow us to run code upon the discovery of an exception.

Another interesting idea is the use of the exception dynamic as a debugging tool. In many cases the application programmer is interested in

situations where the exception is raised. The exception dynamic can contain program code to print information about the context where the exception was raised.

10.5.1 Example 1: exceptions as lists

An example of an alternative exception administration is given below. This administration makes it easier to detect single exceptions out of a set of exceptions. Due to the list nature and the predefined exceptions, we can inspect the set for single elements. Note that the flexibility of the mechanism allows for a user defined exception: **UserExc String**. Note the equality instance for the ExcKind typeclass, where the equality of the exceptions is defined.

Listing 10.9: Example of an alternative administration

```
:: ExcType :== [ExcKind]

:: ExcKind =
MatchFailed | DivByZero | ListOneShorter |
ListTwoShorter | HdFromEmptyList | TlFromEmptyList |
UserExc String

instance == ExcKind
    where
    (==) MatchFailed MatchFailed = True
    (==) DivByZero DivByZero = True
    (==) ListOneShorter ListOneShorter = True
    (==) ListTwoShorter ListTwoShorter = True
    (==) HdFromEmptyList HdFromEmptyList = True
    (==) TlFromEmptyList TlFromEmptyList = True
    (==) (UserExc x) (UserExc y) = x ==y
    (==) _ _ = False

(UN) :: ExcType ExcType -> ExcType
(UN) a b = a ++ b

emptyExc :: ExcType
emptyExc = []
```

```
sCaseOf :: (ST a) [((ST a),(ST b))] (Result b) -> ST
    b
sCaseOf x y z = doSCaseOf [MatchFailed] x y z

sDiv :: (ST ((ST Int) -> ((ST Int) -> ST Int)))
sDiv= doSDiv [DivByZero]

sZip :: (ST ((ST (List (ST a))) -> ((ST (List (ST b)))
    -> ST
(List (ST (a,b))))))
sZip = doSZip [ListOneShorter] [ListTwoShorter]

sHd :: (ST ((ST (List (ST a))) -> ST a))
sHd = doSHd[HdFromEmptyList]

sTl :: (ST ((ST (List a)) -> ST (List a)))
sTl = doSTl[TlFromEmptyList]
```

10.5.2 Example 2: Undo, Retry using exceptions

The next example shows us the great flexibility of our solution. Combined with dynamics this leads to interesting designs. An example is the use of dynamics to store code to restore a pre-condition.

Listing 10.10: Example of undo/retry

```
//The Exception Scheme:
:: ExcType :== [Exc]

:: ExcKind =
MatchFailed | DivByZero | ListOneShorter
| ListTwoShorter | HdFromEmptyList | TlFromEmptyList
| UserExc String

:: Exc = E ExcKind Dynamic // Dynamic contains state

instance == Exc
    where (==) (E x _) (E y _) = x == y

(UN) :: ExcType ExcType -> ExcType (UN) a b = a ++ b
```

```
emptyExc :: ExcType emptyExc = []

//Example functions:
sCaseOf :: (ST a) [((ST a),(ST b))] (Result b) -> ST b
sCaseOf x y z = doSCaseOf [E MatchFailed (dynamic "")]
    x y z

sDiv :: (ST ((ST Int) -> ((ST Int) -> ST Int)))
sDiv = lift sDiv'
    where sDiv' x y = App2 (doSDiv [E DivByZero (
       dynamic (x,y))]) x y

//Example of simulated db connections:
//f :: s -> s; simulates an always succeeding function
f x = App1 sInc x

//g :: s -> s; simulates a sometimes failing function
g x = App2 sDiv sOne x

// just ordinary lists to simulate a stream of
   transactions
// contents is in save domain

//transactions :: [ST (s -> s)]
transactions = map lift [g, f o f o f o f, g, f o g, g
   , g, f o f, f
o f o g]

doTransactions [] _ = []
doTransactions [action : rest] arg
    # newState = App1 action arg
    # msg = sCaseOf (App1 isValid newState)
        [(sTrue, lift "transaction committed")]
        (Default (lift "transaction failed"))
    # nextState = catch always (lift getOldState)
       newState
    = [(nextState, msg) : (doTransactions rest
       nextState)]
    where
    getOldState [E DivByZero ( (_, oldState) :: (a, ST
```

```
        Int))] = cast oldState
                    // cast to force the "unique"
                       restore of old state

// run "series of transactions" and display results
Start = (map (displayExpr o fst) program, (map (
   displayExpr o snd)
program))
    where
    program = doTransactions transactions sZero
```

This example simulates database connections as functions. **transactions** represents a list of database transactions. These functions may, in reality, fail or succeed. Failure in our case is simulated by raising an exception. Using a exception scheme consisting of lists of exception type and dynamic combinations, it is possible to restore a state after an exception occurred.

In **doTransactions** we see a new state is formed using **App1 action arg**, where arg is sZero. If the new state is valid, a transaction succeeded. If the new state is a failed state, i.e. the transaction failed, we restore the previous state as stored in the exception. The function **getOldState** retrieves the stored state from the exception.

The cast in the function **getOldState** is needed to preserve uniqueness. Retrieving a state from a dynamic delivers a non-unique state. The compiler and type checking algorithm cannot determine the uniqueness of the retrieved element. From the context, we know that the state is unique (we put it there ourselves), hence this cast is safe.

Chapter 11

Reflection and future research

In this chapter we investigate the effects of our proposal for current functional programming languages. The chapter starts with a quick summary. We compare our solution to the current solution of Haskell. We finish with a discussion about the direction of future research.

11.1 Achievements of this thesis

We summarize the achievements of this thesis. The following paragraphs each point out a feature of our solution.

11.1.1 Catch without IO Monad

Haskell's catch needs the IO Monad to determine what exception is catched out of a set of exceptions. In contrast, our solution has all exceptions (and possible code in the form of dynamics) in an ExcType. A catch is the inspection of the ErrorDomain (the ExcType) for the type of exceptions you are interested in. The example in listing 10.7 shows an example of a catch without using an IO monad or unique world reference.

11.1.2 Exceptions are no longer values!

Another aspect of exception handling as implemented in Haskell that causes problems is the fact that exceptions are treated as values. At first sight this seems like a logical choice, following the path of reasoning of IEEE

floating point standards[com]. The IEEE standard for floating point calculation defines certain bit-patterns for NaN in case of for example division by zero. This standard was the inspiration for the Haskell design of exceptions. However one function call may throw any number of exceptions. Our solution evades this problem by using a entire error domain to store exceptions. This evades problems as the addition of two or more exceptions and problems comparing the return values of two exceptional functions. The downside of this design discission is that the set of all exceptions is propagated trough each function call. An example of our approach is shown in listing 10.5 of paragraph 10.4.1.

11.1.3 No loss of exception information

Other exception mechanism's for lazy functional languages often loose information about exceptions. Haskell's solution forces the choice of one exception. Our proposal preserves all exceptions.

11.2 Comparing Haskell's exception handling to our proposal

The essential ideas of our solution are the same as Haskell's. Both solutions use an alternate element to indicate exceptions. Where Haskell uses an extension to the value system, we use an error co-domain. Haskell uses only one "value" indicating exceptions due to their semantics, where a single exceptions is regarded as the representation of all exceptions that occurred before a catch.

In our solution we use the union of values and the domain of exceptions, and thus we can show every single exception that occurred before a catch.

Haskell's single exception encoded as value allows them to keep the language lazy. In contrast, our proposed solution needs a global "exception state" to indicate either error domain or normal value domain. We still lack an implementation or formal semantics that prove this global exception state is possible and completely preserves laziness.

11.2.1 Downsides with respect to Haskell's solution

The solution presented in this thesis is very different from regular exception handling in programming languages. The use of an ErrorDomain makes it harder for application programmers to use this technique of error handling. Furthermore no efficient implementation exists. If this design is implemented

in Clean, lots of plumbing code needs to be generated by the compiler. Also one needs a well defined set of functions that clearly define the mechanism for application programmers.

11.2.2 Asynchronous exceptions

In this document we deliberately ignored asynchronous events that generate exceptions. The imprecise semantics of the Haskell solution make it very suitable for dealing with exceptions that are raised in an asynchronous manner.

No research has been conducted to the suitability of our solution for asynchronous exceptions. Application of our design to asynchronous exceptions would at least require further integration into the runtime system of Clean.

11.2.3 Finally construction

The design presented in chapter 9, has no support for a finally construction, as found in for example Java and Haskell.

11.3 The plus problem

An interesting phenomenon exists in exception handling in lazy functional languages. At any point in a function call graph there can be any number of exception from nodes in the subtree. Due to the lazy semantics, the order in which those exceptions were thrown is not defined. Call by need only evaluates function calls when the value of a term is needed. Depending on the implementation, the order of exceptions may be different each run of the program. Subsequent executions don't guarantee the same order of exceptions.

We have shown that Haskell's solution uses the IO monad to choose one exception from a set of exceptions. This choice is based on an "outside oracle", the IO monad. One loses information about the order of exceptions.

Our solution uses a custom exception representation, combining exceptions to some form of collection. It is possible to construct an excType that preserves order.

We now encounter the question of whether or not the order of exceptions is relevant. When an exception is raised, the system is in an erroneous state. Using exception handling, we expose the internal evaluation order of our programming system. However, this exposition is only visible upon

inspection of the collection of exceptions. The problem is somewhat like Schrődinger's cat[Sch]. Inspection of the collection of exceptions introduces the problem of order, whereas the set itself fits within the abstract semantics.

11.4 Topics we did not cover in this thesis

In chapter 6, we named a number of features an exception system should adhere to. One of these features is to keep transformations valid. We did not investigate if our solution keeps transformations valid.

Other topics that have open issues are: research into increased strictness due to inspection of the error domain indicator. Another point of research is a formal semantics for our solution.

11.5 Ideas for future research

As we have shown in this document, the main problems any exception handling mechanism for lazy pure functional programming languages should solve are a way to efficiently represent sets of exceptions and a way to inspect values without losing laziness. Any attempts to design an alternative to either the design presented here or the current Haskell design needs to focus on these two problems.

Furthermore it's important to keep in mind that a design actually is usable for the average programmer. For example, there are often question on haskell IRC[1] channels about catch in exception handling in Haskell. These question almost always have to do with catch being in the IO monad and the inability to see the relation between an IO monad and exception handling.

Another prime area for research is to come up with an efficient implementation of the design described in this document. One of the main areas of attention should be the error domain indicator, showing if a program is in an erroneous state or in a normal state. Inspecting this value at the correct times is paramount for preserving laziness.

Problems like the plus-problem hint at a deeper issues. It is possible solutions to the problem of exception handling in lazy functional languages exist at the level of the fundamental computational model. Future research may focus on extending typed lambda calculus with exceptions in the context of call-by-need evaluation.

[1] An internet chat mechanism

11.5.1 Uniqueness types

Clean uses a special extension to it's type system to support IO and other side-effects. The idea is that if one always has exactly one reference to an object, it is safe to perform IO on this object.

In our design, the state of a program is always either the "normal" value domain or the exceptional "Error" domain. If we take care that the passing of a unique variable is reference preserving, uniqueness typing should be gained "for free". This is still a topic for further research.

We show the type declaration of the catch, raise functions in example 11.1. Note that a * indicates the uniqueness attribute.

Listing 11.1: Type declarations of catch and raise

```
catch :: .(ExcType -> .Bool) .(ST (ExcType -> *(ST a))
    ) u:(ST a) ->
v:(ST a), [u <= v]

raise :: ExcType -> *(ST a)
```

These compiler generated type signatures show us that the ST a type has uniqueness attributes, this suggests our solution works in the presence of uniqueness typing. Further research should be conducted to the integration of uniqueness typing and our exception mechanism.

11.6 Implementation issues

There are still a number of implementation issues. The technique that is often used for exception handling. unwinding the stack, doesn't work for our design. In a "normal" exception handling mechanism, the stack is unwinded (removing stack frames) until the runtime reaches the stack frame pointing to the exception handler.

The flag indicating whether or not we are in the error domain must be implemented in the runtime environment as a bit, to enable efficient switching. Perhaps even per module of code. Besides this, we would like a switch to enable or disable runtime support for the ErrorDomain. This in order to reduce overhead of exceptions where the programmer is sure none will happen. The efficient implementation of our mechanism is still an open issue.

The error domain approach needs modifications to the runtime engine of Clean. The error domain dynamic must be stored somewhere in an efficient manner.

11.7 Speculative remarks

The solution presented in this document is not so much a replacement of exception handling. It's another way of looking towards error handling in lazy functional languages. The continuing problems in the implementation of exception handling in lazy functional problems point towards perhaps fundamental problems mixing imperative like structures as exceptions in a lazy programming language. Most of the current lazy languages with exception handling use either a semantic trick to make things work or do admissions towards referential transparency or transformations. As Simon Peyton Jones writes in his paper "Wearing the hair shirt: a retrospective on Haskell" [Jon03], it may not always be beneficial to use lazy evaluation. Perhaps selectively disabling and combining lazy and strict evaluation can solve these problems. The use of dynamics is still relatively new. It allows us to circumvent many of the limitations of the type system used in Clean.

Chapter 12

Conclusion

This thesis discussed exception handling in lazy functional languages, specifically Clean, from a software engineering perspective.

We described current problems with exception handling. Showed the need for exceptions even though their use implies a penalty in terms of performance or less clean semantics. Finally we presented an alternative to existing exception mechanisms. We started this discussion with a mini language, showing the mechanisms and ideas of our solution. A prototype in Clean was presented, showing the practical use of the design.

Finally we discussed future research and open issues of our solution.

In chapter 3 we stated a number of demands. In this chapter we evaluate whether or not we met our demands.

1. **Usable design** Due to our "exceptions as a collection" approach and extensible exception scheme's our solution offers flexibility for programmers.

2. **Efficient implementation** As of now, there is no efficient implementation available. The hard part is finding a suitable implementation for our exception collection.

3. **Intuitively clear semantics** Our solution offers great flexibility and hence offers user-definable semantics. The user should only adhere to the rules we stated for defining an exception scheme.

4. **Semantically correct** Using our mini language in chapter 9, we showed that a correct semantics is plausible. A formal proof however, is topic for further research.

We did not meet all demands stated in the problem statement. The main contribution of this thesis is the introduction of the idea of the error domain. Further research should point out if this idea leads to an efficient implementation.

Bibliography

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[Bar84] H.P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*. North-Holland, 1984.

[Bar88] H. P. Barendregt. Introduction to lambda calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988.

[CGN04] Richard Carlsson, Bjorn Gustavsson, and Patrik Nyblom. Erlang's exception handling revisited. In *Erlang Workshop '04*. ACM, 2004.

[CH02] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics, 2002.

[com] IEEE committee. IEEE floating point standard. Available on: `http://stevehollasch.com/cgindex/coding/ieeefloat.html`.

[HO89] John Hughes and John O'Donnell. Expressing and reasoning about nondeterministic functional programs. In *Functional Programming, Glasgow 1989*, Workshops in Computing, pages 308–328. Springer-Verlag, 1989.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley Professional, 2003.

[Jon03] Simon Peyton Jones. Wearing the hair shirt: a retrospective on haskell. Invited talk at POPL 2003, 2003.

[JRH+99] Simon L. Peyton Jones, Alastair Reid, Fergus Henderson, C. A. R. Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–36, 1999.

[MJMR01] Simon Marlow, Simon L. Peyton Jones, Andrew Moran, and John H. Reppy. Asynchronous exceptions in haskell. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 274–285, 2001.

[NSvEP91] Eric Nocker, Sjaak Smetsers, Marko van Eekelen, and Rinus Plasmeijer. Concurrent clean. In Leeuwen Aarts and Rem, editors, *Proc. of Parallel Architectures and Languages Europe (PARLE '91)*, volume 505, pages 202–219. Springer-Verlag, 1991.

[PE93] Rinus Plasmeijer and Marko Van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman Publishing Co., Inc., 1993.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[Sch] Erwin Schrődinger. The present situation in quantum mechanics. Available on: `http://www.tu-harburg.de/rzt/rzt/it/QM/cat.html`.

[Str91] Bjarne Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley, 1991.

[Unk] Unknown. Haskell library reference manual. Available on: `http://www.haskell.org/ghc/docs/latest/html/libraries/base/`.

[vdB03] S. van den Berg. A comparison of exception-handling mechanisms. Earlier paper, 2003.

[vWP03] Arjen van Weelden and Rinus Plasmeijer. A functional shell that dynamically combines compiled code. In *Proceedings 15th International Workshop on the Implementation of Functional Languages, IFL 2003, Selected Papers*, 2003.

[Wad92] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM Press.

[wc] Haskell www crew. Haskell WWW site. Available on: `http://www.haskell.org/`.

```
definition module ExcBasis

import StdOverloaded
import ExcScheme

:: ST a = ST Status a ExcType

:: Status = Correct | Error

// catch; args:
// 1: what exceptions are catcheable
// 2: handler: takes ExcType and delivers save value; hence handler may raise ex
ceptions again
// 3: the save value itself

catch :: .(ExcType -> .Bool) .(ST (ExcType -> *(ST a))) u:(ST a) -> v:(ST a), [u
 <= v]
raise :: ExcType -> *(ST a)

// handy for catcheable predicate of catch

always :: .a -> .Bool
never :: .a -> .Bool

// lift any variable to save domain

lift :: .a -> *(ST .a)

// apply of save function (1/2/3/4 args)

App1 :: (ST .(.a -> u:(ST b))) .a -> v:(ST b), [u <= v]
App2 :: (ST .(.a -> .(.b -> u:(ST c)))) .a .b -> v:(ST c), [u <= v]
App3 :: (ST ((ST a) -> ((ST b) -> ((ST c) -> (ST d))))) (ST a) (ST b) (ST c) ->
ST d
App4 :: (ST ((ST a) -> ((ST b) -> ((ST c) -> ((ST d) -> (ST e)))))) (ST a) (ST b
) (ST c) (ST d) -> (ST e)

// case of for save domain

:: Result b = NoDefault | Default (ST b)

// case of; args:
// 1: exception if case fails
// 2: matching value
// 3: list of cases: (case, result)
// 4: default result (if no case matches); may be NoDefault
// see SaveFuns for demo of use

doSCaseOf :: ExcType (ST a) [(ST a, ST b)] (Result b) -> ST b | == a & TC a & TC
 b

// repeat: def of list, for transform to save domain

:: List a = Nil | Cons a (List a)

// save Nil and Cons; sCons is function, so use App2

sNil :: ST (List a)
sCons :: (ST ((ST a) -> ((ST (List (ST a))) -> ST (List (ST a)))))

// handy functions

// make a const save list
```

Appendix A: Prototype

```
listToSaveList :: [(ST a)] -> ST (List (ST a))

// displayExpr(List): just a hack to display the "result" of an expression (list)

displayExpr :: !(ST a) -> b
displayExprList :: (ST (List (ST a))) -> .[b]

// functions that are to be superfluous; catch should suffice

//      isOK :: !(ST a) -> Bool
//      isExc :: !(ST a) -> ST Bool
//      isValid :: !(ST a) -> ST Bool
//      getExc :: !(ST a) -> ExcType
```

```
implementation module ExcBasis

import StdEnv, cast

import StdException

:: Status = Correct | Error

:: ST a = ST Status a ExcType

// catch
// handler itself can produce or be exception
// just propagate if exception is not caught
// note: ExcType may contain set of exceptions, depending on scheme
catch :: .(ExcType -> .Bool) .(ST (ExcType -> *(ST a))) u:(ST a) -> v:(ST a), [u
 <= v]
catch _ _ arg=:(ST Correct _ _)                         = arg
catch catcheable handler (ST Error _ exc)       = if (catcheable exc) (AppToExc
handler exc) (raise exc)

AppToExc :: (ST (ExcType -> *(ST a))) ExcType -> *(ST a)
AppToExc (ST Error _ ex) _                  = raise ex
AppToExc (ST _ handler _) exc   = handler exc

// raise covers also unique variant

raise :: ExcType -> *(ST a)
raise exc = ST Error undef exc

// handy for catcheable predicate of catch

always :: .a -> .Bool
always x = True

never :: .a -> .Bool
never x = False

// lift any variable to save domain

lift :: .a -> *(ST .a)
lift a = ST Correct a emptyExc

// apply of save function

App1 :: (ST .(.a -> u:(ST b))) .a -> v:(ST b), [u <= v]
App1 (ST Error _ ex) _  = raise ex
App1 (ST _ f _) arg     = f arg

// apply of save function with 2/3/4 args

App2 :: (ST .(.a -> .(.b -> u:(ST c)))) .a .b -> v:(ST c), [u <= v]
App2 (ST Error _ ex) _ _        = raise ex
App2 (ST _ f _) arg1 arg2       = f arg1 arg2

App3 :: (ST ((ST a) -> ((ST b) -> ((ST c) -> (ST d))))) (ST a) (ST b) (ST c) ->
ST d
App3 (ST Error _ ex) _ _ _              = raise ex
App3 (ST _ f _) arg1 arg2       arg3    = f arg1 arg2 arg3

App4 :: (ST ((ST a) -> ((ST b) -> ((ST c) -> ((ST d) -> (ST e)))))) (ST a) (ST b
) (ST c) (ST d) -> (ST e)
App4 (ST Error _ ex) _ _ _ _                    = raise ex
App4 (ST _ f _) arg1 arg2       arg3 arg4       = f arg1 arg2 arg3 arg4
```

Appendix A: Prototype

```
// case of for save domain

:: Result b = NoDefault | Default (ST b)

// case of; args:
// 1: exception if case fails
// 2: matching value
// 3: list of cases: (case, result)
// 4: default result (if no case matches); may be NoDefault
// see SaveFuns for demo of use

doSCaseOf :: ExcType (ST a) [(ST a, ST b)] (Result b) -> ST b | == a & TC a & TC
 b
doSCaseOf caseOfExc checkVal listOfCases defaultResult = case checkVal of
        ST Error _ exc          = raise exc // propagate exception
        ST Correct val _        = matchCase val listOfCases
                where
//              matchCase :: a [(ST a, ST b)] -> ST b | == a
                matchCase _ [] = case defaultResult of // check whether there is
 a defaultResult
                        Default x       = x
                        NoDefault       = raise caseOfExc // otherwise raise cas
eOfExc
                matchCase matchVal [(a, result) : restListOfCases] = case a of
                        ST Error _ exc  = raise exc
                        ST Correct a _
                                | val == a      = result
                                | otherwise = matchCase matchVal restListOfCases
 // next el of list of cases

// repeat: def of list, for transform to save domain

:: List a = Nil | Cons a (List a)

// save Nil and Cons; sCons is function

sNil :: ST (List a)
sNil = lift Nil

sCons :: (ST ((ST a) -> ((ST (List (ST a))) -> ST (List (ST a)))))
sCons = lift cons
        where
        cons :: (ST a) (ST (List (ST a))) -> ST (List (ST a))
        cons a (ST Correct x _) = lift (Cons a x)
        cons a (ST Error _ exc) = lift (Cons a (Cons (ST Error undef exc) Nil))

// handy functions

// make a const save list

listToSaveList :: [(ST a)] -> ST (List (ST a))
listToSaveList []               = sNil
listToSaveList [a:x]    = App2 sCons a (listToSaveList x)

// displayExpr(List): just a hack to display nicely the "result" of an expression
(list)

displayExpr :: !(ST a) -> b
displayExpr (ST Correct a _) = cast a
displayExpr (ST Error _ exc) = cast exc

displayExprList :: (ST (List (ST a))) -> .[b]
displayExprList (ST Error _ exc) = [cast exc]
displayExprList (ST Correct Nil _) = []
```

Appendix A: Prototype

```
displayExprList (ST Correct (Cons a x) _) = [displayExpr a : displayExprList (ST
 Correct x emptyExc)]

// functions that are to be superfluous; catch should suffice

// this one for use in (Clean) pattern match

isOK :: !(ST a) -> Bool
isOK (ST Correct _ _)   = True
isOK (ST Error _ _)             = False

isExc :: !(ST a) -> ST Bool
isExc x = case x of
        ST Error _ _    = ST Correct True emptyExc
        ST Correct _ _  = ST Correct False emptyExc

getExc :: !(ST a) -> ExcType
getExc (ST Error _ exc) = exc
```

Appendix A: Prototype

```
definition module ExcPrimFuns

import ExcBasis

// all kinds of functions generating exceptions by themselves

instance == (ST a) | == a
instance == (List a) | == a
instance < (ST a) | < a

doSPrim :: .(.a -> .(.b -> c)) !(ST .a) (ST .b) -> .(ST c)

doSDiv :: ExcType -> (ST u:((ST .Int) -> v:((ST .Int) -> .(ST Int)))), [v <= u]

doSHd :: ExcType -> (ST .((ST (List u:(ST a))) -> v:(ST a))), [u <= v]
doSTl :: ExcType -> (ST .((ST .(List a)) -> .(ST (List a))))

doSZip :: ExcType ExcType -> (ST u:((ST (List .(ST a))) -> v:((ST (List .(ST b))
) -> ST (List (ST (a,b)))))), [v <= u]
```

```
implementation module ExcPrimFuns

import StdEnv
import ExcBasis

// all kinds of functions having access to the internal representation
// or generating exceptions by themselves

instance == (ST a) | == a
        where
        (==) (ST Correct x _) (ST Correct y _) = x == y
        (==) _ _ = False

instance == (List a) | == a
        where
        (==) (Cons a x) (Cons b y) = a == b && x == y
        (==) Nil Nil = True
        (==) _ _ = False

instance < (ST a) | < a
        where
        (<) (ST Correct x _) (ST Correct y _) = x < y
        (<) _ _ = False

doSPrim :: .(.a -> .(.b -> c)) !(ST .a) (ST .b) -> .(ST c)
doSPrim op a b = prim a b
        where
        prim arg1 arg2 = case arg1 of
        ST Error _ exc = raise exc
        ST Correct x _ = case arg2 of
            ST Error _ exc = raise exc
            ST Correct y _ = ST Correct (op x y) emptyExc

doSDiv ::ExcType -> (ST u:((ST .Int) -> v:((ST .Int) -> .(ST Int)))), [v <= u]
doSDiv divByZeroExc = lift div
    where
    div x y = case x of
        ST Correct a _ = case y of
            ST Correct b _ = case b of
                    0 = raise divByZeroExc
                    _ = ST Correct (a/b) emptyExc
            ST Error _ exc = ST Error undef exc
        ST Error _ exc = raise exc

doSHd :: ExcType -> (ST .((ST (List u:(ST a))) -> v:(ST a))), [u <= v]
doSHd emptyListExc = lift hd
        where
        hd (ST Error _ exc)             = raise exc
        hd (ST _ Nil _)                 = raise emptyListExc
        hd (ST _ (Cons a x) _)  = a

doSTl :: ExcType -> (ST .((ST .(List a)) -> .(ST (List a))))
doSTl emptyListExc = lift tl
        where
        tl (ST Error _ exc)             = raise exc
        tl (ST _ Nil _)                 = raise emptyListExc
        tl (ST _ (Cons a x) _)  = lift x

doSZip :: ExcType ExcType -> (ST u:((ST (List .(ST a))) -> v:((ST (List .(ST b))
) -> ST (List (ST (a,b)))))), [v <= u]
doSZip exc1 exc2 = lift zip
        where
        zip (ST Error _ exc) _                  = raise exc
        zip _ (ST Error _ exc)                  = raise exc
```



```
        zip (ST _ Nil _) (ST _ Nil _)   = sNil
        zip (ST _ Nil _) _                              = raise exc1
        zip _ (ST _ Nil _)                              = raise exc2
        zip (ST _ (Cons a x) _) (ST _ (Cons b y) _) = case a of
            ST Correct hd1 _    = case b of
                ST Correct hd2 _        = App2 sCons (ST Correct (hd1, hd2) empt
yExc) (App2 (lift zip) (lift x) (lift y))
                ST Error _ excB         = App2 sCons (raise excB) (App2 (lift zi
p) (lift x) (lift y))
            ST Error _ excA  = case b of
                ST Correct _ _  = App2 sCons (raise excA) (App2 (lift zip) (lift
 x) (lift y))
                ST Error _ excB = App2 sCons (raise (excA UN excB)) (App2 (lift
zip) (lift x) (lift y))
```

Appendix A: Prototype

```
definition module ExcScheme

import ExcBasis, GenEq

// scheme 1: just String's
:: ExcType :== String

// scheme 2: constructors
//:: ExcType :== [ExcKind]

// scheme 3: constructors and args in dynamic
//:: ExcType :== [Exc]

//:: Exc = E ExcKind Dynamic // Dynamic contains state

//instance == Exc

:: ExcKind      = MatchFailed | DivByZero | ListOneShorter | ListTwoShorter
                | HdFromEmptyList | TlFromEmptyList | UserExc String

instance == ExcKind

(UN) :: !ExcType !ExcType -> ExcType

emptyExc :: ExcType

sCaseOf :: .(ST a) [(.(ST a),.(ST b))] .(Result b) -> ST b | == a & TC a & TC b

sDiv :: (ST u:((ST .Int) -> v:((ST .Int) -> ST Int))), [v <= u]

sZip :: (ST u:((ST (List .(ST a))) -> v:((ST (List .(ST b))) -> ST (List (ST (a,
b)))))) | TC a & TC b, [v <= u]
sHd :: (ST .((ST (List .(ST a))) -> ST a)) | TC a
sTl :: (ST .((ST .(List a)) -> ST (List a))) | TC a
```

```
implementation module ExcScheme

import StdEnv
import ExcBasis, ExcPrimFuns

// +++++++++++++++++++++ scheme 1: just String's +++++++++++++++++++++

:: ExcType :== String

(UN) :: !ExcType !ExcType -> ExcType
(UN) a b = a +++ " FOLLOWED BY " +++ b

emptyExc :: ExcType
emptyExc = ""

sCaseOf :: .(ST a) [(.(ST a),.(ST b))] .(Result b) -> ST b | == a & TC a & TC b
sCaseOf x y z  = doSCaseOf "case didn't match" x y z

sDiv :: (ST u:((ST .Int) -> v:((ST .Int) -> ST Int))), [v <= u]
sDiv = doSDiv "exception in sDiv: divide by zero"

sZip :: (ST u:((ST (List .(ST a))) -> v:((ST (List .(ST b))) -> ST (List (ST (a,
b)))))) | TC a & TC b, [v <= u]
sZip = doSZip "sZip: first arg is shorter" "sZip: second arg is shorter"

sHd :: (ST .((ST (List .(ST a))) -> ST a)) | TC a
sHd = doSHd "hd from empty list"

sTl :: (ST .((ST .(List a)) -> ST (List a))) | TC a
sTl = doSTl "tl from empty list"

// +++++++++++++++++++++ scheme 2: constructors +++++++++++++++++++++

//:: ExcType :== [ExcKind]

:: ExcKind      = MatchFailed | DivByZero | ListOneShorter | ListTwoShorter
                | HdFromEmptyList | TlFromEmptyList | UserExc String

instance == ExcKind
        where
        (==) MatchFailed MatchFailed = True
        (==) DivByZero DivByZero = True
        (==) ListOneShorter ListOneShorter = True
        (==) ListTwoShorter ListTwoShorter = True
        (==) HdFromEmptyList HdFromEmptyList = True
        (==) TlFromEmptyList TlFromEmptyList = True
        (==) (UserExc x) (UserExc y) = x ==y
        (==) _ _ = False

//      (UN) :: !ExcType !ExcType -> ExcType
//      (UN) a b = a ++ b
//
//      emptyExc :: ExcType
//      emptyExc = []
//
//      sCaseOf :: .(ST a) [(.(ST a),.(ST b))] .(Result b)  -> ST b | == a & TC
a & TC b
//      sCaseOf x y z = doSCaseOf [MatchFailed] x y z
//
//      sDiv :: (ST u:((ST .Int) -> v:((ST .Int) -> ST Int))), [v <= u]
//      sDiv = doSDiv [DivByZero]
//
//      sZip :: (ST u:((ST (List .(ST a))) -> v:((ST (List .(ST b))) -> ST (List
 (ST (a,b)))))) | TC a & TC b, [v <= u]
```



```
//      sZip = doSZip [ListOneShorter] [ListTwoShorter]
//
//      sHd :: (ST .((ST (List .(ST a))) -> ST a)) | TC a
//      sHd = doSHd [HdFromEmptyList]
//
//      sTl :: (ST .((ST .(List a)) -> ST (List a))) | TC a
//      sTl = doSTl [TlFromEmptyList]

// +++++++++++++++++++++ scheme 3: constructors and args in dynamic ++++++++++++
+++++++++

//      :: ExcType :== [Exc]
//
//      :: Exc = E ExcKind Dynamic // Dynamic contains state
//
//      instance == Exc
//              where (==) (E x _) (E y _) = x == y
//
//      (UN) :: !ExcType !ExcType -> ExcType
//      (UN) a b = a ++ b
//
//      emptyExc :: ExcType
//      emptyExc = []
//
//      sCaseOf :: .(ST a) [(.(ST a),.(ST b))] .(Result b) -> ST b | == a & TC a
 & TC b
//      sCaseOf x y z = doSCaseOf [E MatchFailed (dynamic "")] x y z // no trace

//
//      sDiv :: (ST u:((ST .Int) -> v:((ST .Int) -> ST Int))), [v <= u]
//      sDiv = lift sDiv'
//              where sDiv' x y = App2 (doSDiv [E DivByZero (dynamic (x,y))]) x
y
//
//      sZip :: (ST u:((ST (List .(ST a))) -> v:((ST (List .(ST b))) -> ST (List
 (ST (a,b)))))) | TC a & TC b, [v <= u]
//      sZip = lift sZip'
//              where sZip' x y = App2 (doSZip [E ListOneShorter (dynamic (x,y))
] [E ListTwoShorter (dynamic (x,y))]) x y
//
//      sHd :: (ST .((ST (List .(ST a))) -> ST a)) | TC a
//      sHd = lift sHd'
//              where sHd' x = App1 (doSHd [E HdFromEmptyList (dynamic x)]) x
//
//      sTl :: (ST .((ST .(List a)) -> ST (List a))) | TC a
//      sTl = lift sTl'
//              where sTl' x = App1 (doSTl [E TlFromEmptyList (dynamic x)] ) x
```

Appendix A: Prototype

```
definition module SafeFuns

// all kinds of funs which have to be redefined because of different type ST ...

// once the use of the co-domain is implemented in the compiler, this is not lon
ger necessary

import StdEnv
import ExcBasis

instance + (ST a) | + a
instance - (ST a) | - a
instance * (ST a) | * a
instance / (ST Int)

isValid :: (ST .(.(ST a) -> ST Bool)) | == a & TC a

sInc :: (ST ((ST Int) -> ST Int))
sDivBy :: (ST ((ST Int) -> ST Int))

sIsEven :: (ST ((ST Int) -> ST Bool))
sCmp :: (ST ((ST a) -> ((ST a) -> (ST Bool)))) | < a

sMap :: .(ST ((ST ((ST a) -> .(ST b))) -> (ST (List (ST a))) -> ST (List (ST b))
)) | == a & TC a & TC b
sFilter :: .(ST ((ST ((ST a) -> (ST .Bool))) -> (ST (List (ST a))) -> ST (List (
ST a)))) | == a & TC a
sSum :: .(ST ((ST (List (ST Int))) -> ST Int))

// sCatchingSum ignores exceptions (gives them a value 0)
sCatchingSum :: .(ST ((ST (List (ST Int))) -> ST Int))

sMinOne     :: *(ST Int)
sZero       :: *(ST Int)
sOne        :: *(ST Int)
sTwo            :: *(ST Int)
sTen        :: *(ST Int)
sOneHundred :: *(ST Int)
sTrue       :: *(ST Bool)
sFalse      :: *(ST Bool)
kermit      :: *(ST String)
missPiggy   :: *(ST String)
cook        :: *(ST String)
```

Appendix A: Prototype

```
implementation module SafeFuns

import StdEnv
import ExcBasis, ExcPrimFuns

// all save funs take the form
// f = lift f' where f' =
// these save funs only use App1/App2/.. for apply and sCaseOf for pattern match


isValid :: (ST .(.(ST a) -> ST Bool)) | == a & TC a
isValid = lift isValid'
        where
        isValid' x = catch always (lift (\x -> sFalse)) (sCaseOf x [] (Default s
True))

instance + (ST a) | + a where (+) arg1 arg2 = doSPrim (+) arg1 arg2
instance - (ST a) | - a where (-) arg1 arg2 = doSPrim (-) arg1 arg2
instance * (ST a) | * a where (*) arg1 arg2 = doSPrim (*) arg1 arg2
instance / (ST Int) where (/) arg1 arg2 = App2 sDiv arg1 arg2

sInc :: (ST ((ST Int) -> ST Int))
sInc = lift sInc'
        where sInc' = (+) sOne

sDivBy :: (ST ((ST Int) -> ST Int))
sDivBy = lift sDivBy'
        where sDivBy' = (/) sOne

sIsEven :: (ST ((ST Int) -> ST Bool))
sIsEven = lift sIsEven'
        where
        sIsEven' :: (ST Int) -> ST Bool
        sIsEven' a = sCaseOf ((a/sTwo)*sTwo)
                [(a,sTrue)]
                (Default sFalse)

sCmp :: (ST ((ST a) -> ((ST a) -> (ST Bool)))) | < a
sCmp = lift sCmp'
        where
        sCmp' a b = lift (a < b)


sMap :: .(ST ((ST ((ST a) -> .(ST b))) -> (ST (List (ST a))) -> ST (List (ST b)
)) | == a & TC a & TC b
sMap = lift sMap'
        where
        sMap' sF sList = sCaseOf sList
                [(sNil, sNil)]
                (Default (App2 sCons (App1 sF (App1 sHd sList)) (App2 sMap sF (A
pp1 sTl sList))))

sFilter :: .(ST ((ST ((ST a) -> (ST .Bool))) -> (ST (List (ST a))) -> ST (List (
ST a)))) | == a & TC a
sFilter = lift sFilter'
        where
        sFilter' sPred sList = sCaseOf sList
                [(sNil, sNil)]
                (Default checkHd)
                where
                checkHd = sCaseOf pred
                        [(sTrue, App2 sCons hd (App2 sFilter sPred tl))]
                        (Default (App2 sFilter sPred tl))
                pred = App1 sPred hd
```


Appendix A: Prototype

```
                hd = App1 sHd sList
                tl = App1 sTl sList

sSum :: .(ST ((ST (List (ST Int))) -> ST Int))
sSum = lift sSum'
        where
        sSum' :: (ST (List (ST Int))) -> ST Int
        sSum' sList = sCaseOf sList
                [(sNil, sZero)]
                (Default (App1 sHd sList + App1 sSum (App1 sTl sList)))

// sCatchingSum ignores exceptions (gives them a value 0)

sCatchingSum :: .(ST ((ST (List (ST Int))) -> ST Int))
sCatchingSum = lift sCatchingSum'
        where
        sCatchingSum' :: (ST (List (ST Int))) -> ST Int
        sCatchingSum' sList = sCaseOf sList
                [(sNil, sZero)]
                (Default result)
                where
                result = catch always (lift (\x -> sZero)) (App1 sHd sList) + Ap
p1 sCatchingSum (App1 sTl sList)

sMinOne     :: *(ST Int)
sMinOne     = lift (-1)

sZero       :: *(ST Int)
sZero       = lift 0

sOne        :: *(ST Int)
sOne        = lift 1

sTwo       :: *(ST Int)
sTwo       = lift 2

sTen        :: *(ST Int)
sTen        = lift 10

sOneHundred :: *(ST Int)
sOneHundred = lift 100

sTrue       :: *(ST Bool)
sTrue       = lift True

sFalse      :: *(ST Bool)
sFalse      = lift False

kermit      :: *(ST String)
kermit      = lift "Kermit"

missPiggy   :: *(ST String)
missPiggy   = lift "Miss Piggy"

cook        :: *(ST String)
cook        = lift "Swedish Cook"
```

Appendix A: Prototype

```
definition module StdException

import // everything of exceptions
                ExcScheme
        ,       ExcBasis
        ,       ExcPrimFuns
        ,       SafeFuns
```

Appendix A: Prototype

```
implementation module StdException
```

Appendix A: Prototype

```
module start1

// demonstrates use of exceptions
// use scheme 1, 2 (or 3)

import StdException

Start =
    (   displayExpr sDivByZero1                          // exception in sDiv: di
vide by zero
    ,   displayExpr catched                                      // 100
    ,   displayExpr (App1 sDivBy sZero)         // exception in sDiv: divide by
zero
        ,       displayExpr sDivByZero2                          // exception in
sDiv: divide by zero
        ,       displayExpr sDivByZero3                          // exception in
sDiv: divide by zero
        ,       displayExpr sum
    )
    where
    sDivByZero1 = sOne / sZero
    catched = catch always (lift (\x -> sOneHundred)) sDivByZero1
    sDivByZero2 =  sOne * (App1 sInc sDivByZero1)
    sDivByZero3 = App1 sInc sDivByZero1 - sOne
    sum = App1 sCatchingSum myList
    myList = listToSaveList [sZero, sOne/sZero, sOne, sTen, sOneHundred]
```

Appendix A: Prototype

```
module start2

// demonstrates use of lazy lists with exceptions
// use scheme 1 or scheme 2

import StdException

theDBConns :: ST (List (ST (String, Int)))   // name, connection time; time out
simulated by divide by zero
theDBConns = App2 sZip theSNameList errIntList

theSNameList :: ST (List (ST String))
theSNameList = listToSaveList [kermit, missPiggy, cook]

errIntList :: ST (List (ST Int))
errIntList = App2 sMap (lift divFun) theSIntList
        where divFun = App2 sDiv sOneHundred

theSIntList :: ST (List (ST Int))
theSIntList = listToSaveList [sZero, sOne, sTen]

// theValidDBConns: all not "broken" DB Connections
theValidDBConns :: ST (List (ST (String, Int)))
theValidDBConns = App2 sFilter isValid theDBConns

Start = displayExprList theValidDBConns
```

Appendix A: Prototype

```
module start3

// demonstrates restore of old state if transaction failed somehow
// use scheme 3

import cast
import StdException

//f :: s -> s; simulates an always succeeding function
f x = App1 sInc x

//g :: s -> s; simulates a sometimes failing function
g x = App2 sDiv sOne x

// just ordinary lists to simulate a stream of transactions
// contents is in save domain

//transactions :: [ST (s -> s)]
transactions = map lift [g, f o f o f o f, g, f o g, g, g, f o f, f o f o g]

doTransactions [] _ = []
doTransactions [action : rest] arg
        # newState = App1 action arg
        # msg = sCaseOf (App1 isValid newState)
                [(sTrue, lift "transaction committed")]
                (Default  (lift "transaction failed"))
        # nextState = catch always (lift getOldState) newState
        = [(nextState, msg) : (doTransactions rest nextState)]
        where
        getOldState [E DivByZero ( (_, oldState) :: (a, ST Int))] = cast oldStat
e
                                        // cast to force the "unique" restore of
old state

// run "series of transactions" and display results
Start = (map (displayExpr o fst) program, (map (displayExpr o snd) program))
        where
        program = doTransactions transactions sZero
```

Appendix A: Prototype

Table of Contents