Imprecise Exceptions - Exceptions in Haskell

Christopher Krauß

Universität des Saarlandes
Informatik

22nd March 2006

Outline

- Introduction
- Exceptions
  - Where do we use exceptions?
  - Different kinds of exceptions
  - Problems with pure and lazy languages
  - How to represent exceptions in a lazy language?
- Present a new design based on sets of exceptions to model imprecision
- Sketch a semantics for this design
- Some extensions of the basic idea

Introduction

Imprecise exceptions at the hardware level:

- Modern super scalar microprocessors
- Many instructions run in parallel (increasing performance)
- First exception encountered might not be the first encountered in a sequential run

Use this idea at the programming level:

- Improving performance by changing evaluation order
- May change which exception is encountered first
- Solving this problem: trade precision for performance
- Present a design in `Haskell` depending on the `IO` monad

Exceptions

Where do we use exceptions?

- Disaster recovery
- Alternative result
- Short circuit control flow
- Asynchronous events

Kinds of exceptions:

- Synchronous exceptions
- Asynchronous exceptions

Exceptions in a lazy language

Why are exceptions not available in pure and lazy languages?

- Lazy evaluation scrambles control flow
  => Programs do not have a readily predictable control flow
- Purity is violated if exceptions are used in the usual way
- Exceptions as values

Exceptions as values

- `data ExVal a = OK a | Bad Exception`
- Good things about this approach:
  - No extension to the language is necessary
  - Type indicates whether the function can raise an exception
  - Impossible to forget to handle an exception
  - `ExVal` forms a *monad* $\Rightarrow$ Comfortable use
- Problems with this approach
  - Increased strictness
  - Excessive clutter:
    - Exceptions do not propagate implicitly
    - Inefficient
  - Loss of modularity and reuse of code
  - Loss of transformations

Goals of the new design

- For programs that don't invoke exceptions:
  Unchanged semantics and unaffected efficiency
- All useful transformations remain valid
- Possibility to reason about the exceptions a program might raise
- Stay lazy and keep referential transparency

Basic Idea

- Keep the idea of exceptions as values,
  not as control flow (lazy evaluation!)
- Extend this idea:
  A value of any type is *normal* or *exceptional*
- ```
  data Exception = DivideByZero
                 | Overflow
                 | UserError String
                 | ...
  ```
- `raise :: Exception -> a`
- `catch :: a -> ExVal a`
  ```

Propagation

- Automatic propagation
- But think of laziness:

```
zipWith f [] [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f   xs     ys   = raise UserError "Uneq lists"
```

- Exceptions may be hidden in partially evaluated term
- Propagation only if evaluation is forced ($\neq$ ML)

Catching exceptions

- `catch :: a -> ExVal a`
- But what about:
  `catch ((1/0) + (raise Overflow))`
  Which exception is delivered?

Catching exceptions

- `catch ::  a -> ExVal a`
- But what about:
  `catch ((1/0) + (raise Overflow))`
  Which exception is delivered?
- Possible solutions to this problem:

Catching exceptions

- `catch :: a -> ExVal a`
- But what about:
  `catch ((1/0) + (raise Overflow))`
  Which exception is delivered?
- Possible solutions to this problem:
  - Fix the evaluation order
    → Violates laziness

Catching exceptions

- `catch :: a -> ExVal a`
- But what about:
  `catch ((1/0) + (raise Overflow))`
  Which exception is delivered?
- Possible solutions to this problem:
  - Fix the evaluation order
    $\longrightarrow$ Violates laziness
  - Go non-deterministic
    $\longrightarrow$ Violates purity and referential transparency (e.g. β-reduction)

Catching exceptions

- `catch ::  a -> ExVal a`
- But what about:
  `catch ((1/0) + (raise Overflow))`
  Which exception is delivered?
- Possible solutions to this problem:
  - Fix the evaluation order
    $\longrightarrow$ Violates laziness
  - Go non-deterministic
    $\longrightarrow$ Violates purity and referential transparency (e.g. β-reduction)
  - Return *both* exceptions:
    Exceptional values contain a set of exceptions
    $\longrightarrow$ Implementation has to keep track of the whole set
    $\longrightarrow$ Propagation not automated
    $\longrightarrow$ Violates laziness

Fixing `catch`

- Denotational: Think of maintaining the whole set
- Operational: Stay imprecise, choose one member of the set

Get rid of the non-determinism problem:

- Put `catch` into the `IO` monad:
  `catch :: a -> IO (ExVal a)`
- `IO t` is a computation which
  - Is evaluated without side effects
  - Does only have an effect when it is performed
- Each call to `catch` can make a different choice
- Purity and referential transparency remain
- **Non-determinism in exceptions separated from non-determinism in values**

Relation between denotational and operational semantics

- Difference between denotational and operational semantics
- Difference not visible in pure subset (observed by denotational semantics)
- Performing the `IO` monad denotationally not covered
- Exceptions are not observable in the pure part of the language

Semantics of the design

- $$[e_1 + e_2]\rho = \begin{cases} v_1 + v_2 & \text{if } OK\ v_1 = [e_1]\rho \\ & \text{and } OK\ v_2 = [e_2]\rho \\ Bad(\mathcal{S}[e_1]\rho \cup \mathcal{S}[e_2]\rho) & \text{otherwise} \end{cases}$$
- But what about:
  `loop + raise Overflow`
- Model $\perp$ as follows:
  $\perp \;=\; \mathcal{E} \;\cup\; \{\texttt{NonTermination}\}$
- Straight forward rules for constants, variables, `raise`, abstractions, applications, constructors, and `fix`
- Slightly more complicated for case to maintain transitions

Semantics of catch

- $$\begin{array}{lcl} \texttt{catch}\ (OK\ \ v) & \longrightarrow & \texttt{return}\ (\texttt{OK}\ v) \\ \texttt{catch}\ (Bad\ \ s) & \longrightarrow & \texttt{return}\ (\texttt{Bad}\ x) \\ & \text{if} & x \in s \\ \texttt{catch}\ (Bad\ \ s) & \longrightarrow & \texttt{catch}\ (Bad\ \ s) \\ & \text{if} & \texttt{NonTermination} \in s \end{array}$$
- In our example `loop + raise Overflow`:
  Return any exception or non-termination are valid reactions

Implementation

- Standard exception handling mechanism
- `catch` forces the evaluation of its argument to head normal form
- Evaluation of `raise ex` trims the stack to the top most `catch` mark and returns `Bad ex`
- `catch` returns `OK val` if there is no exception
- Efficiency of programs that do not invoke exceptions stays unaffected
- Exceptional value behaves as first class value

Extensions

- Asynchronous exception (every transition can cause an exception)
- Detectable bottoms - detectable divergence
- Pure functions on exceptional values:
  - Possible to compute on exceptional values
    `mapException :: (Exception -> Exception) -> a -> a`
  - Not possible to return from exceptional to normal values
    - `catch` is non-deterministic
    - `isException :: a -> Bool`:
      `isException loop`
      Consider `isException ((1/0) + loop)`

Other Languages

- Design less expressive than in other languages
- In ML:
  - Declare exceptions locally
  - Raise and handle it without being visible from the outside
- `IO` monad like a trap door
- But no loss of useful transformations

Summary

- All useful transformations stay valid
  (Transformations use program equivalences)
- Some equivalences get lost:
  `error "a" = error "b"` no longer holds
  → Some transformations are refined
- Scales to other extensions, such as adding concurrency
- Model used in Glasgow Haskell compiler (4.0 and later)

References

- S. P. Jones, A. Reid, T. Hoare, S.Marlow, Fergus Henderson. A semantics for imprecise exceptions. *PLDI'99 Atlanta*.
- S. P. Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Microsoft Research, Cambridge* 23rd May 2005.
- S. P. Jones, S. Marlow, A. Moran, and J.Reppy. Asynchronous exceptions in Haskell. *PLDI 2000*.
- S. Thompson. Haskell: The Craft of Functional Programming. *International Computer Science Series*. 1996.

What about case

- `case x of (a,b) -> case y of (p,q) -> e`
  `=`
  `case y of (p,q) -> case x of (a,b) -> e`
  should hold
-

$$
\begin{array}{lll}
[\texttt{case}\ e\ \texttt{of}\ \{p_i \rightarrow r_i\}]\rho & & \\
\quad = [r_i]\rho[v/p_i] & \text{if} & \mathit{OK}\ v = [e]\rho \\
 & \text{and} & v \text{ matches } p_i \\
\quad = \mathit{Bad}(s \cup (\bigcup_i \mathcal{S}([r_i]\rho[\mathit{Bad}\{\}/p_i]) & \text{if} & \mathit{Bad}\ s = [e]\rho
\end{array}
$$

β-reduction

```
let x = (1/0) + (raise Overflow)
in catch x = catch x
```