

Errors in Haskell

- `let average xs = div (sum xs) (length xs)`
`in average []`
- `let tail (x : xs) = xs`
`in tail []`
- `if ... then error "some string" else ...`
- `let loop = loop`
`in loop`

Traditionell, oft alle Fehlerursachen unter „ \perp “ subsumiert.

Besser, feinere Unterscheidung. Etwa wie folgt:

Ok v : nicht fehlerbehaftet

Bad “...” : endlich fehlerbehaftet

\perp : nicht terminierend

Fortpflanzung von Fehlern

- `tail [1/0, 2.5] ~> Ok ((Ok 2.5) : (Ok []))`
- `(λx → 3) (error "...") ~> Ok 3`
- `(error s) (...)` ~> *Bad s*
- `case (error s) of {...}` ~> *Bad s*
- `(error s1) + (error s2) ~> ???`

Abhängigkeit von Auswertungsreihenfolge führt zu erheblichen Einschränkungen der Implementationsfreiheit!

Imprecise Error Semantics [Peyton Jones et al. 1999]

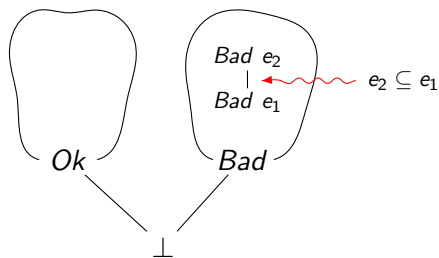
Grundidee:

Ok v : nicht fehlerbehaftet

Bad $\{\dots\}$: endlich fehlerbehaftet, nichtdeterministisch

\perp : nicht terminierend

Definiertheits-Ordnung:



Imprecise Error Semantics [Peyton Jones et al. 1999]

Fortpflanzung von Fehlern:

- $(\text{error } s_1) + (\text{error } s_2) \rightsquigarrow \text{Bad } \{s_1, s_2\}$
- $3 + (\text{error } s) \rightsquigarrow \text{Bad } \{s\}$
- $\text{loop} + (\text{error } s) \rightsquigarrow \perp$
- $(\text{error } s_1) (\text{error } s_2) \rightsquigarrow \text{Bad } \{s_1, s_2\}$
- $(\lambda x \rightarrow 3) (\text{error } s) \rightsquigarrow \text{Ok } 3$
- $\text{case } (\text{error } s_1) \text{ of } \{(x, y) \rightarrow \text{error } s_2\} \rightsquigarrow \text{Bad } \{s_1, s_2\}$

Auswirkungen auf Programmäquivalenz

„Normalerweise“:

$$\text{takeWhile } p (\text{map } h \ l) = \text{map } h (\text{takeWhile } (p \circ h) \ l)$$

wobei:

$$\text{takeWhile} :: (\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$$
$$\text{takeWhile } p \ [] = []$$
$$\text{takeWhile } p \ (a : as) \begin{cases} | p \ a & = a : \text{takeWhile } p \ as \\ | \text{otherwise} & = [] \end{cases}$$
$$\text{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$$
$$\text{map } h \ [] = []$$
$$\text{map } h \ (a : as) = h \ a : \text{map } h \ as$$

Aber nun:

$$\text{takeWhile } \text{null} \ (\text{map } \text{tail} \ (\text{error } s)) \quad \not\equiv s$$
$$\text{map } \text{tail} \ (\text{takeWhile } (\text{null} \circ \text{tail}) \ (\text{error } s)) \quad \not\equiv s \text{ oder } \not\equiv \text{“empty list”}$$

Auswirkungen auf Programmäquivalenz

Denn:

$$\text{takeWhile } (\text{null} \circ \text{tail}) (\text{error } s) \rightsquigarrow \text{Bad } \{s, \text{"empty list"}\}$$

wobei:

$$\begin{aligned} \text{takeWhile } p [] &= [] \\ \text{takeWhile } p (a : as) & \begin{cases} | \text{ } p \ a & = a : (\text{takeWhile } p \ as) \\ | \text{ otherwise} & = [] \end{cases} \end{aligned}$$
$$\begin{aligned} \text{tail } [] &= \text{error "empty list"} \\ \text{tail } (a : as) &= as \end{aligned}$$
$$\begin{aligned} \text{null } [] &= \text{True} \\ \text{null } (a : as) &= \text{False} \end{aligned}$$

Auswirkungen auf Programmäquivalenz

Denn:

`takeWhile (null ◦ tail) (error s) ∼∼ Bad {s, "empty list"}`

während:

`takeWhile null (map tail (error s)) ∼∼ Bad {s}`

wobei:

$$\begin{array}{l} \text{takeWhile } p \ [] \quad = \ [] \\ \text{takeWhile } p \ (a : as) \mid \begin{array}{l} p \ a \\ \text{otherwise} \end{array} \quad = \ a : (\text{takeWhile } p \ as) \end{array}$$
$$\begin{array}{l} \text{map } h \ [] \quad = \ [] \\ \text{map } h \ (a : as) = \ (h \ a) : (\text{map } h \ as) \end{array}$$
$$\begin{array}{l} \text{null } [] \quad = \ \text{True} \\ \text{null } (a : as) = \ \text{False} \end{array}$$

Auswirkungen auf Programmäquivalenz

Denn:

```
takeWhile (null ◦ tail) (error s)  $\rightsquigarrow$  Bad {s, "empty list"}
```

während:

```
takeWhile null (map tail (error s))  $\rightsquigarrow$  Bad {s}
```

Also:

```
takeWhile null (map tail (error s))  
     $\neq$   
map tail (takeWhile (null ◦ tail) (error s))
```

Man stelle sich dies in folgendem Kontext vor:

```
catchJust errorCalls (evaluate ...)  
    ( $\lambda s \rightarrow$  if  $s ==$  "empty list"  
              then return [[42]]  
              else return [])
```


Freies Theorem

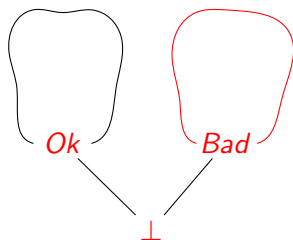
Bisheriger Kenntnisstand:

$$g \ p \ (\text{map } h \ l) = \text{map } h \ (g \ (p \circ h) \ l)$$

für jedes $g :: (\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$, wenn

- $p \neq \perp$,
- h strikt ($h \ \perp = \perp$) und
- h total ($\forall x \neq \perp. h \ x \neq \perp$).

Was sind entsprechende Bedingungen „in echt“?



Schweiß und Tränen ...

- ... durchgängige Formalisierung der Semantik
- ... Einstieg in Beweis des Parametritäts-Theorems
- ... Betrachtung der interessanten Induktionsfälle
- ... Identifizierung geeigneter Bedingungen auf Relationsebene
- ... Anpassung relationaler Aktionen
- ... Vervollständigung allgemeiner Beweis
- ... Übertragung der Bedingungen auf Funktionsebene
- ... Anwendung auf konkrete Funktionen

... Anwendung auf „takeWhile“

Für jedes $g :: (\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$,

$$g \ p \ (\text{map } h \ l) = \text{map } h \ (g \ (p \circ h) \ l)$$

vorausgesetzt

- p und h nicht fehlerbehaftet,
- $h \ \perp = \perp$,
- h verhält sich als Identität auf endlichen Fehlern, und
- h bildet Nichtfehler auf Nichtfehler ab.

