型輪講
14章 Exceptions

尾上 浩一

平成15年5月20日

1 はじめに

本章では例外に関して取り扱う．

プログラム中で例外を発生する状況としては以下のような場合が考えられる．

- 0による割り算
- 算術演算によるオーバーフロー
- 不正なインデックスを用いた配列のアクセス
- 指定したファイルが存在しない，開けない
- メモリの不足　　等

例外を扱かえるようにするためには例外処理を発生させる機構，例外を処理する機構が必要である．例外は呼び出し側で取り扱うのではなく，例外ハンドラによって取り扱いが行われるようにする．本章では以下の流れで展開する．

- 例外が発生したとき全てのプログラムを中止する機構を定義する．
- 例外の捕捉，例外からの回復をするための機構を定義する．
- 例外通知と例外処理の間で付加的な情報を取り扱える機構を定義する．

2 例外の発生

まず最初に，error項を導入し，例外を $\lambda_{\rightarrow}$ で取り扱うための規則を図1に示す．

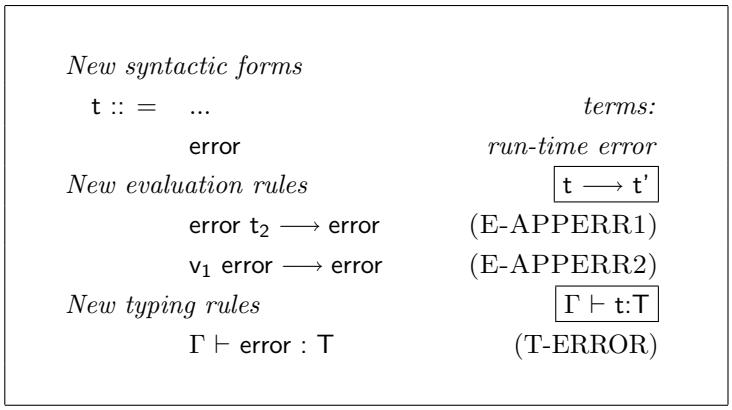

図 1: Errors

図 1 で定義されている規則の注意点としては error が *value* に含まれないということである．これにより E-APPABS と E-APPERR2 間で曖昧性が生じない．(評価関係は決定的である．)

例: (λx:Nat.0) error

error が *value* でないので，E-APPABS は適用されず，E-APPERR2 が適用され，error に評価される．

例: (fix(λx:Nat.x)) error

E-APPERR2 より，左側の項が *value* であるときに error に評価されるので，上のような項は発散する．

型付け規則ではどのようなコンテキストの中でも例外を扱えるようにしたいので， error は任意の型をとれるようにしている．(T-ERROR) この規則は柔軟性はあるが，" すべての型付け可能な項はユニークな型である"(Theorem 9.3.3) という性質を壊してしまうため，型検証アルゴリズムの実装が困難になる．

この解決方法としてプログラマによって各々のコンテキスト内で明示的に error に適切な型付けをするという方法が考えられるが，以下のような場合にうまくいかない．

例: (λ x:Nat.x) ((λ y:Bool.5) (error as Bool))

⟶ (λ x:Nat.x) (error as Bool) (下線部を評価)

評価前の項は well-typed な項であるが，1 ステップ評価した後の項は ill-typed な項になってしまう．(preservation が成立しない)

一般的な解決方法として以下のようなものがあり，多くの可能な型をとることができる error の型を一意に定めることができる．

- *subtyping* を導入することにより (15 章参照)， error に最小の型 Bot を割り当てる．
- *parametric polymorphism* を導入することにより（23 章参照），error に

多相型 ∀ X.X を割り当てる．

例外を導入しても preservation の性質はこれまでの章で述べられてきたものと同じである．

ある項が型 T を持ち，1 ステップ評価されたとき，評価後の項も型 T を持つ

progress の性質に関しては少し修正する必要がある．それはこの章で新たに導入した項 error が *value* でない正規形だからである．以下に修正した progress を示す．

t が閉じた well-typed な正規形であるならば，そのとき t は *value* または error である．

以後の例外に関する議論において，この progress, preservation の性質が保たれる．

3 例外処理

error の評価規則は” 巻き戻し呼び出しスタック” とみなすことができる．呼び出しスタックは活性レコード (各関数に対して一つ) の集合で構成されている．図 1 の規則の場合，例外が発生したとき，呼び出しスタックの活性レコードを空になるまでポップし，これまでの全ての処理を無効にする．現在の例外が導入された，多くの言語では呼び出しスタック中に例外ハンドラを配置することができる．例外が発生した場合，最も内側でその例外に対応したハンドラに処理が移る．(ハンドラの処理後，プログラムが継続される場合もある．)

例外処理ハンドラを導入したときの評価・型付け規則を表 2 に示す.

4 値を伝搬する例外

これまでの規則 (図 1，2) により，$\lambda_{\rightarrow}$ に例外の機構が導入された．ここでは例外が発生したときに，例外ハンドラに値を渡せるように規則を拡張する．これにより柔軟な例外処理を行うことができるようになる．この概念を反映した規則を図 3 に示す。

図 3 の型付け規則中の t_{exn} は例外ハンドラに伝搬される値の型である．これに関しては次の節で議論する．ここでの規則では error がコンストラクタ raise

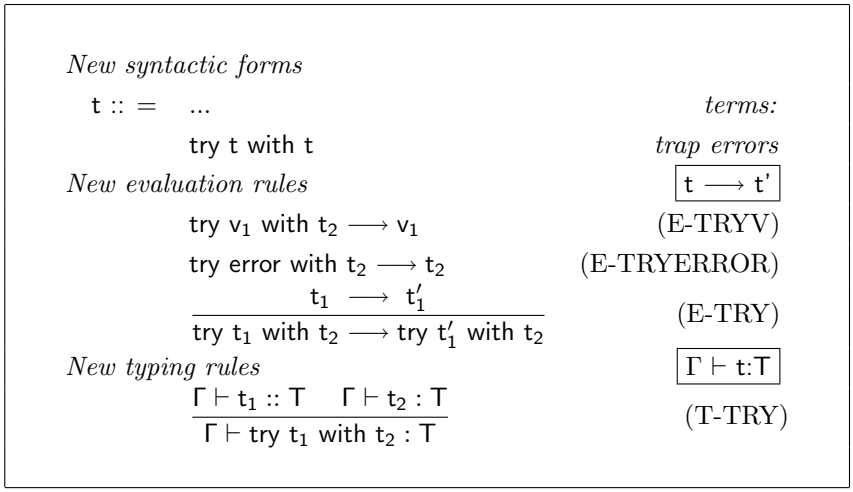

図 2: Error handling

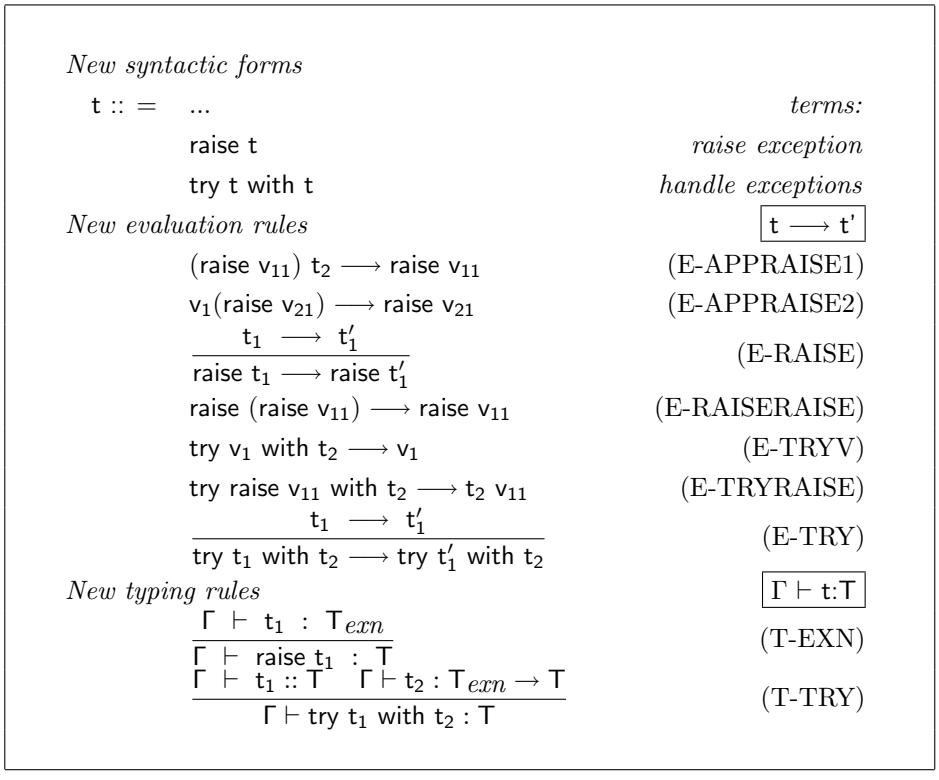

図 3: Exceptions carrying values

t により置き換えられる．[1] ここで注意すべきことは評価規則 E-TRYRAISE において raise の引数が *value* のときのみ例外ハンドラの伝搬が許されるということである．そうでない場合，評価規則 E-RAISE を利用し，*value* まで評価を行う．

5 例外ハンドラに渡される値の型

本章の最後に前節で導入された型 T_{exn} について議論する．

1. T_{exn} に Nat を用いる．UNIX 系の OS で採用されている．システムコールが数字のエラーコードを返す．(エラーが生じないときは 0 を返す)[2]

2. T_{exn} に String を用いる．1 のときに必要なテーブル検索[3]を必要とせず, より叙述的なメッセージを出すことが可能となる．しかし，文字列の構文解析が必要となるかもしれない．

3. T_{exn} に以下のような variant 型を用いる．

   T_{exn} =
   <divideByZero: Unit,
   overflow: Unit,
   fileNotFound: String,
   fileNotReadable: String,
   ... >

   例外ハンドラ中の case で variant 型を利用して異なる種類の例外の処理が可能となる．(異なる伝搬される値の型の処理も可能となる) しかし，あらかじめ例外で利用する全ての variant 型のタグの集合を固定しなければならないため柔軟性に欠ける．

4. T_{exn} に 3 にユーザ定義の例外を取り扱えるようにした拡張 variant 型を用いる．これは ML(exn) で採用されている．

   - ML における例外の宣言

     exception l of T

     l はその時点までに t_{exn} 中に設定されているものと異なるタグで，以後 t_{exn} を $< \mathsf{l}_1 : \mathsf{T}_1 ..... \mathsf{l}_n : \mathsf{T}_n$, l:T> とする．

   - ML における例外発生に関する構文

     raise l(t) $\stackrel{\text{def}}{=}$ raise (<l=t> as T_{exn})

     l は現時点で定義されている例外タグ，t は例外ハンドラに伝搬される値．

[1] t は伝搬される値
[2] 各数字がどのようなエラーであるのかは errno.h を参照.
[3] 各数字がどのようなエラーに対応しているのか

- try 構文

  try t with l(x) → h $\stackrel{\text{def}}{=}$ try t with
  λ e:T_{exn}.case e of
  <l=x> ⇒ h
  | _ ⇒ raise e

  発生した例外 (e) が例外タグ (l) と一致しているか検証する，一致している場合，変数 (x) に伝搬する値をバインドし，例外ハンドラ (h) を評価する．一致しない場合，再び例外を発生させ，対応したハンドラによって発生した例外が処理されるか，またはプログラムのトップレベルに到達するまで例外が伝搬し続ける．

5. T_{exn} にユーザ定義の例外の取り扱いが可能なクラス (Throwable) を用いる．これは Java で採用されている．(throw ⇔ raise, try ... catch ⇔ try ... with) ユーザ定義の例外の宣言は新たに Throwable のサブクラスを定義することにより可能となる．

   Java の例外は ML の例外とは以下の点で異なる．

   - 例外タグに関してサブクラスによる順序付けにより，半順序関係を導入している．これによりあるクラス c1 の例外ハンドラは c1 に関するクラスまたはサブクラスにより発生した例外を取り扱うことが可能となる．
   - 以下の例外に関するクラスを区別する．
     - **Exception**
       プログラムの処理の回復が可能な例外を取り扱う．try ... catch, throw, throws により明示的に発生した例外を処理しなければならない．
     - **Error**
       プログラムの処理の回復が困難または不可能なシステム上の致命的な例外を取り扱う．明示的に例外処理を行う必要がない．