`nea`, `neatop` - nML Exception Analyzer

Sukyoung Ryu

August 31, 2001

1 Introduction

nML Exception Analyzer (EA) analyzes a given nML program and estimates may-uncaught exceptions for each function in the program.

- Batch analyzer (`nea`)
  - Inputs are nML files and outputs are analysis results.
    eg. `% nea file.n`
  - In case of incremental analysis, two options should be provided. For nML type checker, every included directory should be mentioned by "`-I <dir>`". And for nML EA, all the previously analyzed file names should be listed by "`-F <file>`". The order of previously analyzed file names does not matter.
    eg. `% nea -I kb -F kb/kb1.n kb/kb2.n`
  - Options

`-I <dir>`	Add `<dir>` to the list of include directories searched for typechecked file (`.nty`).
`-F <file>`	Add `<file>` to the list of files searched for pre-analyzed ea file (`.nea`).
`-d`	Print debugging information.
`-p`	Print absyn.
`-t`	Print time profile information.

- Toplevel analyzer (`neatop`)
  Input to the toplevel can span several lines. It terminates by `;;` (a double-semicolon).
  eg.
```
% neatop
nML Exception Analyzer, Version 0.9, 08/06/2001, Sukyoung Ryu [ROPAS/KAIST]
#
```

- Analysis results
  For each function of the program, nML EA reports a set of pairs <exn, loc> of an exception exn and its birth (construction, not declaration) place loc. For example,
  - `emitPar (type/e2m.n:862.14-121)`
    `<NotImplemented, type/e2m.n:58.40-54>`

tells that function `emitPar` defined at "`type/e2m.n:862.14-121`" may have an uncaught exception `NotImplemented`, which is constructed at "`type/e2m.n:58.40-54`."

– `/unnamed ftn/ (type/e2m.n:962.21-40)`

  tells that the unnamed function at "`type/e2m.n:962.21-40`" has no uncaught exception.

– `/emitBlock _ / (type/e2m.n:857.18-258)`
  `<NotSafe, type/e2m.n:58.40-54>`
  tells that curried function `emitBlock` (of type `t -> t' -> t''`), given one argument, returns a function which may have an uncaught exception `NotSafe` when it is called.

– `/handler ftn/ (ppc/ppcEm2Ppc.n:3.9-112)`

  tells that handler function at "`ppc/ppcEm2Ppc.n:3.9-112`" has no uncaught exception. Handler function is the match expression in handle expression: e handle match. If handler's match is not exhaustive, then the analysis says that the handler function (match) has uncaught exceptions.

There are two restrictions to use nML EA: (1) nML compiler 0.91 (n/nml-0.91) and nML Control Flow Analyzer (CFA) should be installed to use nML EA, and (2) every file should be type-checked.

2 Implementation

2.1 Code Structure

The whole code structure of nML EA is shown in Figure 1.

2.2 `Makefile`

A user needs to edit `Makefile`'s `ROOT` and `CFA` directories. They should be the root directory of his/her nML compiler 0.91 and CFA, respectively.
eg. `ROOT=/home/puppy/n/nml-0.91/`
`CFA=/home/puppy/n/nml-0.91/tools/ncfa/`

2.3 `patcheck.n`

`is_exhaustive : (Absyn.pat list) list -> bool` checks whether a given pattern matrix is exhaustive or not.

2.4 `eaperv.n`

This file includes the exception information of the pervasive functions. `get: fid -> exninfo` returns may-uncaught exceptions for a given pervasive function. Currently, nML EA reports "OCaml_Exception" for all the exceptions from OCaml libraries.

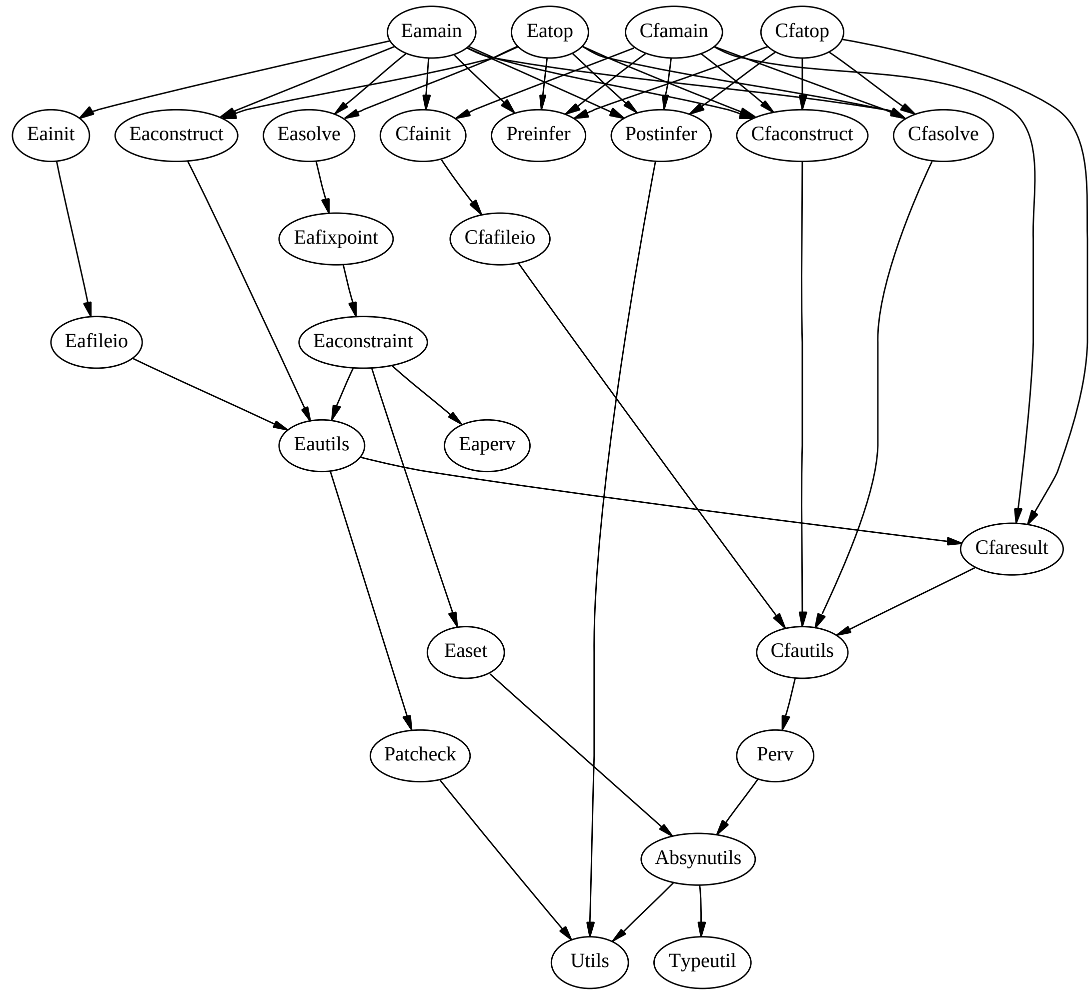

Figure 1: Code Structure of nML EA

When a programmer wants to get more detailed reports, it can be easily extended: the programmer needs to add more information into the `get` function with respect to the `fid` in `perv.n` of nML CFA.

2.5 `eautils.n`

This file includes various utilities for nML EA.

- `fid`, `nFid`, `mkFids`, `mkFids'`: utilities for function id
- `exn_env`, `own_env`, `env`: environments keeping static scoping information during the analysis
- `ePops`, `ePush`, `ePushs`, `oPops`, `oPush`, `oPushs`, `var2exn`, `getOwner`, `mkOwn`, `mkOwn'`: utilities for environments
- `tables`: exception flow equations

- `addEx`, `getEx`, `addP`, `addP'`, `getP`, `addX`, `getX`, `addPP`, `getPP`, `addPX`, `getPX`, `addXP`, `getXP`, `addPR`, `getPR`, `addMR`, `getMR`, `addApp`, `getApp`, `addNfn`, `getNfn`, `copyTbls`, `getTbls`, `setTbls`, `addTbls`, `minusTbls`, `substTbls`, `prTbls`, `prTbls'`: utilities for tables
- contexts keeping analysis results
  - `type basis = {fc: fc, gc: gc, c: ctxt}`
    When analysis results are dumped for the future uses (in case of the incremental analysis), nML EA dumps `basis`. `basis` includes a functor context `fc`, a signature context `gc`, and a structure context `c`.
  - `structure FC : H where type key = StringInfo.fctidinfo`
    `type fc = (ctxt' * ctxt * tables) FC.t`
    A functor context `fc` is a map from functor names to a triple of a functor argument context `ctxt'`, a functor body context `ctxt`, and the analysis results of the functor body `tables`.
  - `structure GC : H where type key = StringInfo.sigidinfo`
    `type gc = ctxt' GC.t`
    `and ctxt' = Ct of gc * cfainfo`
    A signature context `gc` is a map from signature names to its context `ctxt'`. `ctxt'` includes a signature context for its subsignatures and `cfainfo` of the signature body.
  - `structure SC : H where type key = StringInfo.stridinfo`
    `type sc = ctxt SC.t`
    `and ctxt = C of sc * cfainfo`
    A structure context `sc` is a map from structure names to its context `ctxt`. `ctxt` includes a structure context for its substructures and `cfainfo` of the structure body.
  - `type eainfo = {exs: exn_env}`
    `eainfo` includes information of the declared exceptions (`exn_env`).
- `c'2c`, `sids2c`, `sids2c'`, `addFC`, `addGC`, `addSC`, `addEI`, `addC`, `addC'`, `copyCUC`, `copyCUC'`, `prC`, `prC'`: utilities for contexts
- `initEa`, `initEaFid`, `initEaConstruct`, `backward`: initializations
- `unhandled: Absyn.rule list -> (string list * string list * bool) option`
  Given an exception rule list, `unhandled` returns the following result:
  - `None` : The rule list is exhaustive.
    (It is exhaustive iff it has `wildpat` or `varpat`.)
  - `Some (minus, plus, normal)` : The rule list is not exhaustive.
    - `minus` : exceptions with no arguments and exceptions with non-exception-having arguments
    - `plus` : exceptions with non-exception-having arguments whose argument patterns are not exhaustive

[VAR$_\triangleright$] $$f \triangleright x:\ \{X_f \supseteq var(x)\}$$

[C$_\triangleright$] $$f \triangleright \mathtt{1}:\ \emptyset$$

[ABS$_\triangleright$] $$\frac{g \triangleright e_1:\ \mathcal{C}_1}{f \triangleright \lambda_g x_\tau.e_1:\ \mathcal{C}_1}$$

[FIX$_\triangleright$] $$\frac{g \triangleright e_1:\ \mathcal{C}_1 \qquad f \triangleright e_2:\ \mathcal{C}_2}{f \triangleright \mathtt{fix}\, g\, \lambda_g x_\tau.e_1\ \mathtt{in}\, e_2:\ \mathcal{C}_1 \cup \mathcal{C}_2}$$

[EXN$_\triangleright$] $$\frac{f \triangleright e_1:\ \mathcal{C}_1}{f \triangleright \mathtt{exn}\, \kappa\, e_1:\ \{X_f \supseteq \kappa\} \cup \mathcal{C}_1}$$

[DCON$_\triangleright$] $$\frac{f \triangleright e_1:\ \mathcal{C}_1}{f \triangleright \mathtt{decon}\, e_1:\ \mathcal{C}_1}$$

[APP$_\triangleright$] $$\frac{f \triangleright e_1:\ \mathcal{C}_1 \qquad f \triangleright e_2:\ \mathcal{C}_2}{f \triangleright e_1\, e_2:\ \{X_f \supseteq app_X(e_1, X_f), P_f \supseteq app_P(e_1, X_f)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[CASE$_\triangleright$] $$\frac{f \triangleright e_1:\ \mathcal{C}_1 \qquad f \triangleright e_2:\ \mathcal{C}_2 \qquad f \triangleright e_3:\ \mathcal{C}_3}{f \triangleright \mathtt{case}\, e_1\, \kappa\, e_2\, e_3:\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}}$$

[RS$_\triangleright$] $$\frac{f \triangleright e_1:\ \mathcal{C}_1}{f \triangleright \mathtt{raise}\, e_1:\ \{P_f \supseteq X_f\} \cup \mathcal{C}_1}$$

[-RS$_\triangleright$] $$\frac{f \triangleright e_1:\ \mathcal{C}_1}{f \triangleright \mathtt{-raise}\, e_1\, \kappa_1 \cdots \kappa_n:\ \{P_f \supseteq X_f \setminus_{e_1} \{\kappa_1, \cdots, \kappa_n\}\} \cup \mathcal{C}_1}$$

[+RS$_\triangleright$] $$\frac{f \triangleright e_1:\ \mathcal{C}_1}{f \triangleright \mathtt{+raise}\, e_1\, \kappa:\ \{P_f \supseteq X_f \cap_{e_1} \{\kappa\}\} \cup \mathcal{C}_1}$$

[HNDL$_\triangleright$] $$\frac{g \triangleright e_g:\ \mathcal{C}_1 \qquad h \triangleright e_2:\ \mathcal{C}_2}{f \triangleright \mathtt{handle}\, e_g\, \lambda_h x_\tau.e_2:\quad \begin{array}{l}\{X_f \supseteq app_X(\lambda_h x_\tau.e_2, P_g) \cup X_g,\\ P_f \supseteq app_P(\lambda_h x_\tau.e_2, P_g)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

Figure 2: Exception Analysis Rules

* `normal`: whether there exist exceptions with exception-having arguments
* if `normal` then `([], [], true)` else `(minus, plus, false)`

2.6 `eaconstruct.n` and `easolve.n`

`EaConstruct.construct: Absyn.topdec -> CfaUtils.basis -> EaUtils.basis` constructs exception flow equations for a given `absyn` program. And `EaSolve.solve: unit -> EaSet.set array * int` solves the constructed equations. These files implement the exception analysis rules in Figure 2 ([1, 2]).

2.7 `easet.n`

This file encodes an exception information as a unique integer and includes operations (`mkelmt`, `one`, `cup`, `cap`, `subset`, `minus`, `add`, `iter`, `empty`, `prElmt`, and `prSet`) for a set of exceptions.

2.8 `eaconstraint.n`

This file constructs set constraints from the exception flow equations. The syntax of its set expression is as follows:

```
type se =
    VAR of var
  | CONST of EaSet.element
  | CUP of se * se
  | CAP of se * string list
  | MINUS of se * string list
  | EMPTY
```

2.9 `eafixpoint.n`

`tabulate: Cs.var -> EaSet.set array` solves the constructed set constraints by using a worklist algorithm [3]. An input value is the kick-off/root set variable.

2.10 `earesult.n`

`prEa: EaSet.set array -> int -> unit` reports nML EA results.

2.11 `eatop.n` and `eamain.n`

`eatop.n` is a driver for the toplevel analyzer (`neatop`) and `eamain.n` is a driver for the batch analyzer (`nea`). The whole process of nML EA in the nML Compiler 0.91 is shown in Figure 3.

2.12 `eafileio.n`

This file includes the ways to dump and load nML EA results.

- `exception Invalid_header`
  When a dumped file is written by a different version of nML EA, this exception is raised.

- `fileIOext: string`
  The file extension of nML EA is "`.nea`".

- `dump: string -> EaUtils.basis -> unit`
  `dump filename.n result` dumps the analysis result (`result`) of the nML file (`filename.n`) to the result file (`filename.nea`).

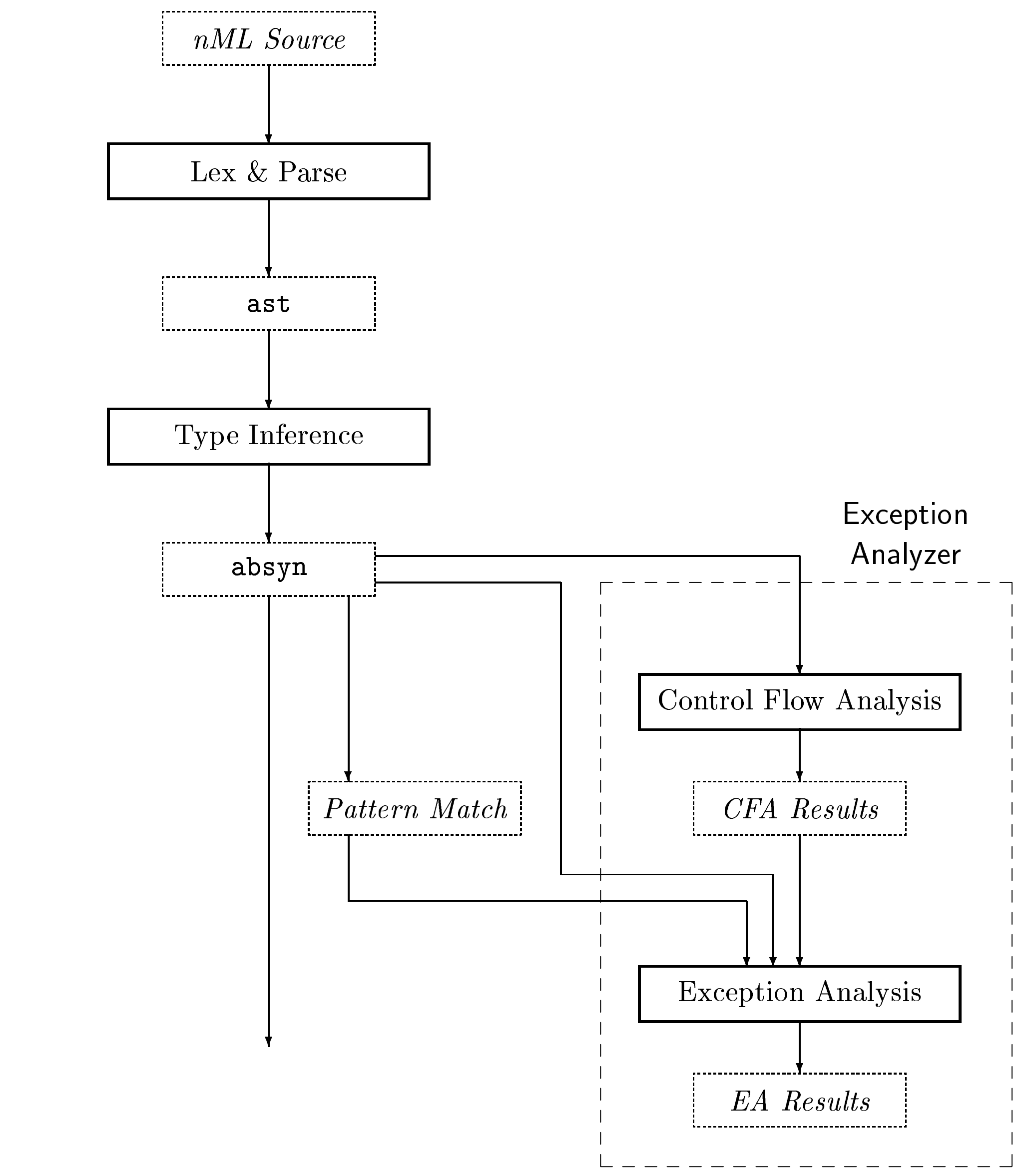

Figure 3: nML Exception Analyzer in the nML Compiler 0.91

- `load: string -> EaUtils.basis * EaUtils.tables`
  `load filename.nfa` loads the pre-analyzed file (`filename.nea`) and returns the pre-analyzed results.

2.13 `eainit.n`

`init fid files` reads all the pre-analyzed files (`files`) and initializes the function id to `fid`.

3 Availability

- You can get nML EA by using cvs at `n/nea`.

References

[1] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Proceedings of the 4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 98–113, Paris, France, September 1997. Springer-Verlag.

[2] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science*, 273(1), 2001. Extended version of [1].

[3] Liling Chen, Luddy Harrison, and Kwangkeun Yi. Efficient computation of fixpoints that arise in complex program analysis. *Journal of Programming Languages*, 3(1):31–68, 1995.