# `ncfa`, `ncfatop` - nML Control Flow Analyzer

Sukyoung Ryu

August 31, 2001

## 1 Introduction

nML Control Flow Analyzer (CFA) analyzes a given nML program and estimates a possible set of function values for each function typed expression.

- Batch analyzer (`ncfa`)
  - Inputs are nML files and outputs are analysis results.
    eg. `% ncfa 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 CFA, all the previously analyzed file names should be listed by "`-F <file>`". The order of previously analyzed file names does not matter.
    eg. `% ncfa -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 cfa file (`.nfa`). |
| `-d` | Print debugging information. |
| `-p` | Print absyn. |
| `-t` | Print time profile information. |

- Toplevel analyzer (`ncfatop`)
  Input to the toplevel can span several lines. It terminates by `;;` (a double-semicolon).
  eg.
  ```
  % ncfatop
  nML Control Flow Analyzer, Version 0.9, 08/06/2001, Sukyoung Ryu [ROPAS/KAIST]
  #
  ```
- Analysis results
  For each location of "$e_1$" in the function application "$e_1\ e_2$" of the program, nML CFA reports a set of pairs <ftn, loc> of a function name ftn and its declaration place loc. For example,
  - `(type/e2m.n:862.14-121)`
    `<emitPar, type/e2m.n:58.40-54>`
    tells that a function-typed expression at "`type/e2m.n:862.14-121`" may be evaluated to the function `emitPar`, which is declared at "`type/e2m.n:58.40-54`."

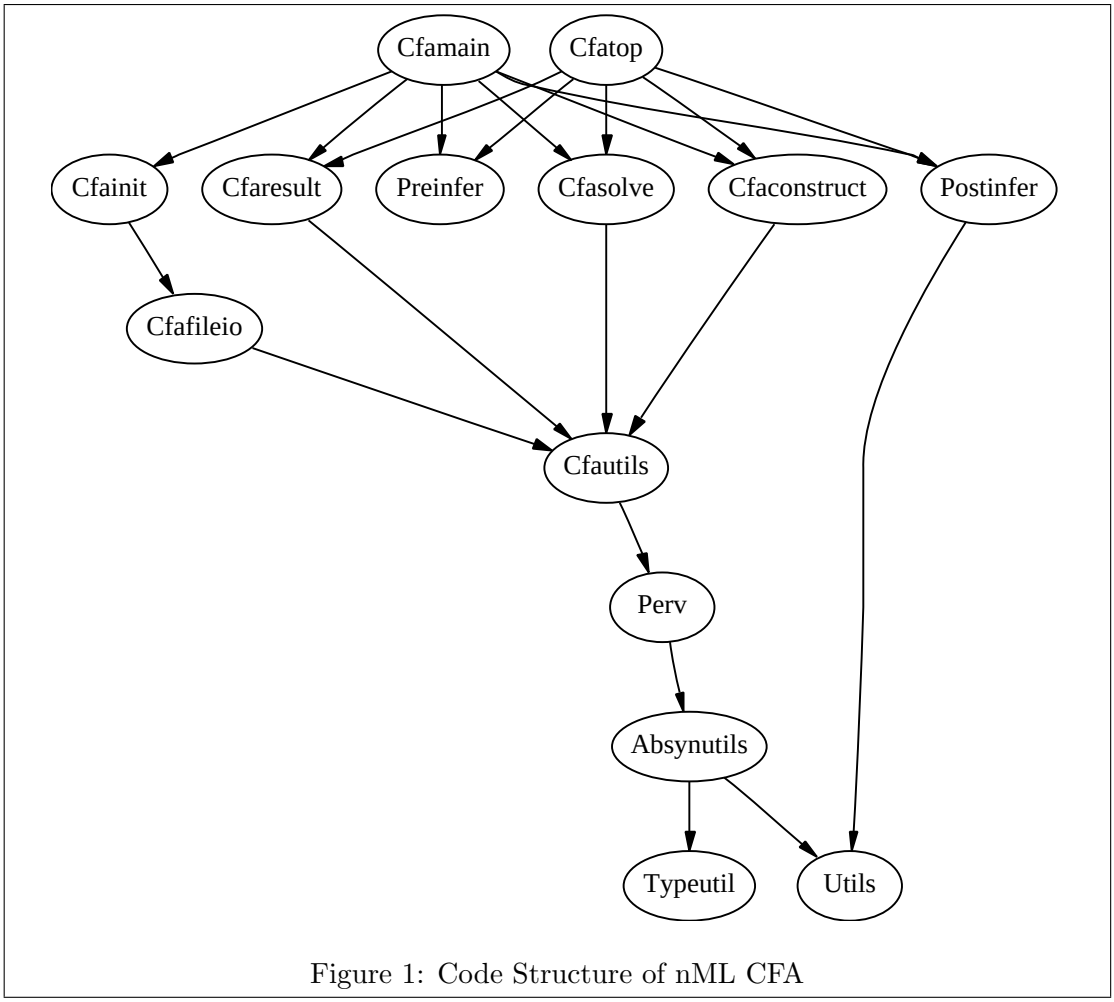

Figure 1: Code Structure of nML CFA

– `(type/e2m.n:962.21-40)`
  `</unnamed ftn/, type/e2m.n:59.11-20>`
  tells that the unnamed function at "`type/e2m.n:962.21-40`" may be called at "`type/e2m.n:962.21-40`".
– `(type/e2m.n:857.18-258)`
  `</emitBlock _ /, type/e2m.n:58.40-54>`
  tells that curried function `emitBlock` (of type `t -> t' -> t''`), given one argument, returns a function which may be called at "`type/e2m.n:58.40-54`".

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

# 2 Implementation

## 2.1 Code Structure

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

### 2.2 `Makefile`

A user needs to edit `Makefile`'s `ROOT` directory. It should be the root directory of his/her nML compiler 0.91.

eg. `ROOT=/home/puppy/n/nml-0.91/`

### 2.3 `utils.n`

This file includes various utility functions for nML CFA and nML Exception Analyzer (EA).

- structures
  - `structure IS`: integer set
  - `structure SS`: string set
  - `structure LS`: location set
  - `structure ES`: exception set
  - `structure IM`: integer map
  - `structure SM`: string map
  - `structure TM`: type map
  - `structure LM`: location map
  - `structure EM`: exception map
  - `structure SH`: string heap
  - `structure T`: for time profile information
- list operations
  - `list2IS: int list -> IS.t`
  - `foldl: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
  - `take: ('a list * int) -> 'a list`
    `take (l,i)` returns the first `i` elements of the list `l`.
  - `drop: ('a list * int) -> 'a list`
    `drop (l,i)` returns what is left after dropping the first `i` elements of the list `l`.
  - `lstMinus: 'a list -> 'a list -> 'a list`
    `lstMinus [3,6,2,4,6,9] [4,6,9]` is `[3,6,2]`.
  - `lstlast: 'a list -> 'a list -> 'a list * 'a`
- others
  - `exception Fail of string`
  - `inc: int ref -> unit`: increment function
  - `($$): ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`: function composition
  - `print: string -> unit`: print_string

- `i2s: int -> string`: integer to string
- `b2s: bool -> string`: boolean to string
- `r2s: real -> string`: real to string
- `cfa: bool ref`: flag for reporting cfa results
- `debugflag: bool ref`: flag for debugging
- `debug: string -> unit`: print function for debugging

### 2.4 preinfer.n

`PreInfer.preinfer: String_ast.Ast.topdec -> String_ast.Ast.topdec` simplifies value bindings in a given `ast` program in order to help the accuracy of the analysis. Simplifications are as follows:

| | | |
|---|---|---|
| `val k a = k` $e$ | $\Rightarrow$ | `val a =` $e$ |
| `val {a, b}={a=`$e_1$`, b=`$e_2$`}` | $\Rightarrow$ | `val a =` $e_1$ `and b =` $e_2$ |
| `val ref a = ref` $e$ | $\Rightarrow$ | `val a =` $e$ |
| `val (a, b) = (`$e_1$`,` $e_2$`)` | $\Rightarrow$ | `val a =` $e_1$ `and b =` $e_2$ |
| `val [a, b] = [`$e_1$`,` $e_2$`]` | $\Rightarrow$ | `val a =` $e_1$ `and b =` $e_2$ |
| `val [\|a, b\|]=[\|`$e_1$`,` $e_2$`\|]` | $\Rightarrow$ | `val a =` $e_1$ `and b =` $e_2$ |
| `val p = (`$e_1$`;` $e_2$`;` $e_3$`)` | $\Rightarrow$ | `val _ = (`$e_1$`;` $e_2$`) and p =` $e_3$ |

### 2.5 postinfer.n

`PostInfer.postinfer: Absyn.topdec -> Absyn.topdec` removes redundant language features in a given `absyn` program in order to help the performance of the analysis.

- It removes `ConstraintPat`, `ConstraintExp` and `ConstraintSig`.
- It transforms `[SimpleDec(`$d_1$`,`$i_1$`), SimpleDec(`$d_2$`,`$i_2$`)]` to `SimpleDec(SeqDec([`$d_1$`,`$d_2$`],`$i_1$`),`$i_2$`)`.

### 2.6 absynutils.n

This file includes the following utility functions for nML `absyn`:

- `varName: Absyn.varid -> string`
- `varLName: Absyn.varlongid -> string`
- `strLName: Absyn.strlongid -> string`
- `conLName: Absyn.conlongid -> string`
- `printTy: Ty.ty -> unit` Print a given type.
- `isFnty: Ty.ty -> bool`
  Whether an input type is a function type.

- `hasFnty: Ty.ty -> bool`
  Whether an input type has a function type.
- `isExnty: Ty.ty -> bool`
  Whether an input type is an exception type.
- `isExn: Ty.ty -> bool`
  Whether an input type is an exception.
- `hasExnty: Ty.ty -> bool`
  whether an input type has an exception type.
- `ioTy: Ty.ty -> bool * bool`
  `ioTy t` is a tuple of whether the input type of `t` has an exception type and whether the output type of `t` has an exception type.
- `appTy: Ty.ty -> int -> Ty.ty`
  `appTy t n` is a resulting type after applying arguments `n` times to the type `t`.
- `outTy: Ty.ty -> int -> bool * bool`
  `outTy t n` is a tuple of whether `appTy t n` is a function type and whether it is an exception type.
- `clearTy: Ty.ty -> Ty.ty`
  Generalize a given type to a type scheme.
- `unifiable: Ty.ty -> Ty.ty -> bool`
  Whether given two types are unifiable.
- `getLoc: LocationInfo.info -> Location.t`
  Get the location from a given `info`.
- `patName: Absyn.pat -> string`
  Get the name of a given pattern.
- `init: unit -> unit`
  Initialize for nML CFA.
- `dummyInfo: LocationInfo.info`
- `loc2str: Location.t -> string`
  Get the string representing a given location.

### 2.7 `perv.n`

This file includes information for pervasive functions and library functions.

- `isPervasives: Absyn.exp -> bool`
  Whether a given expression is a pervasive function or a library function.
- `getPrimitives: string -> fid option`
  Find the fid for a given primitive functions.

- `getPervasives: string -> fid option`
  Find the fid for a given pervasive function or a library function.
- `getAri: fid -> fid`
  Find the arity of a given function fid.

### 2.8 `cfautils.n`

This file includes various utilities for nML CFA.

- `fid`, `nFid`, `mkFids`, `mkFids'`: utilities for function id
- `lexp`: selected source information for nML CFA
- `lid`, `nLid`, `mkLids`: utilities for lexp id
- `ftn_env`, `own_env`, `env`: environments keeping static scoping information during the analysis
- `fPops`, `fPush`, `fPushs`, `oPops`, `oPush`, `oPushs`, `f2f`, `fnm2fid`, `getOwner`, `mkOwn`: utilities for environments
- `tables`: control flow equations
- `count`, `flag`, `addFids`, `addFids'`, `getFids`, `getAri`, `addF`, `getF`, `addT`, `addLa`, `getLa`, `addLb`, `addE`, `addAb`, `addApAb`, `addApL`, `addDmAp`, `lid2le`, `getLid2le`, `addLoc2lid`, `loc2lid`, `addFctApp`, `getFctApp`, `prLoc2lid`, `prF`, `getHifn`, `getTbls`, `setTbls`, `addTbls`, `minusTbls`, `substTbls`, `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 CFA 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 cfainfo = {ff:fid, lf:fid, fl:lid, ll:lid, fns:ftn_env}`
    `cfainfo` includes the range of function id (`ff`, `lf`) and lexp id (`fl`, `ll`) and information of the declared functions (`ftn_env`).

- `c'2c`, `sids2c`, `sids2c'`, `addFC`, `addGC`, `addSC`, `addCI`, `addC`, `addC'`, `copyC`, `copyC'`, `prC`, `prC'`: utilities for contexts
- `initCfa`, `initCfaConstruct`, `initCfaSolve`: initializations
- `fidsbytys`: functions unifiable with a given function type
- `cfasE`, `cfasAp`, `cfasLb`: solving control flow equations
- `result: ((Location.t * lid) list) ref`:
  a list of function position and its lexp id of each function application

### 2.9 `cfaconstruct.n` and `cfasolve.n`

`CfaConstruct.construct: Absyn.topdec -> CfaUtils.basis` constructs control flow equations for a given `absyn` program. And `CfaSolve.solve: unit -> unit` solves the constructed equations. These files implement the call-graph estimation rules in Figure 2 ([1, 2]).

### 2.10 `cfaresult.n`

This file includes several ways to get nML CFA results.

- `getCfa: Location.t -> (string * Location.t) list`
  for general uses: `getCfa` reports a list of the tuples of a function name and its location for a given location
- `getCfaEa1: Location.t -> (CfaUtils.fid * bool * bool) list`
  for nML Exception Analyzer
- `getCfaEa2: CfaUtils.fid -> CfaUtils.fid list`
  for nML Exception Analyzer
- `prCfa: (Format.formatter -> Location.t -> unit) -> unit`
  for reporting nML CFA results

### 2.11 `cfatop.n` and `cfamain.n`

`cfatop.n` is a driver for the toplevel analyzer (`ncfatop`) and `cfamain.n` is a driver for the batch analyzer (`ncfa`). The whole process of nML CFA in the nML Compiler 0.91 is shown in Figure 3.

$$\lambda x.e \to \lambda x.e$$

$$\frac{\texttt{fix}\, f\, \lambda x.e_1 \in \wp}{f \to \lambda x.e_1} \qquad \frac{e_2 \to \lambda y.e}{\texttt{fix}\, f\, \lambda x.e_1\, \texttt{in}\, e_2 \to \lambda y.e}$$

$$\frac{e_1\, e_2 \in \wp, \quad e_1 \to \lambda x.e_1', \quad e_2 \to \lambda y.e}{x \to \lambda y.e}$$

$$\frac{e_1 \to \lambda x.e_1', \quad e_1' \to \lambda y.e}{e_1\, e_2 \to \lambda y.e}$$

$$\frac{e_2 \to \lambda x.e}{\texttt{case}\, e_1\, \kappa\, e_2\, e_3 \to \lambda x.e} \qquad \frac{e_3 \to \lambda x.e}{\texttt{case}\, e_1\, \kappa\, e_2\, e_3 \to \lambda x.e}$$

$$\frac{e_1 \to \lambda y.e}{\texttt{handle}\, e_1\, \lambda x.e_2 \to \lambda y.e} \qquad \frac{e_2 \to \lambda y.e}{\texttt{handle}\, e_1\, \lambda x.e_2 \to \lambda y.e}$$

$$\frac{\mathit{type}(e_1) = \mathit{type}(\texttt{decon}\, e), \quad e_1 \to \lambda x.e'}{\texttt{decon}\, e \to \lambda x.e'}$$

Figure 2: Call-graph Estimation Rules

### 2.12 `cfafileio.n`

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

- `exception Invalid_header`
  When a dumped file is written by a different version of nML CFA, this exception is raised.
- `fileIOext: string`
  The file extension of nML CFA is "`.nfa`".
- `dump: string -> CfaUtils.basis -> unit`
  `dump filename.n result` dumps the analysis result (`result`) of the nML file (`filename.n`) to the result file (`filename.nfa`).
- `load: string -> CfaUtils.basis * CfaUtils.tables`
  `load filename.nfa` loads the pre-analyzed file (`filename.nfa`) and returns the pre-analyzed results.

### 2.13 `cfainit.n`

`init: string list -> CfaUtils.fid * string list` reads all the pre-analyzed files and returns a pair of the number of pre-analyzed functions and the ordered name

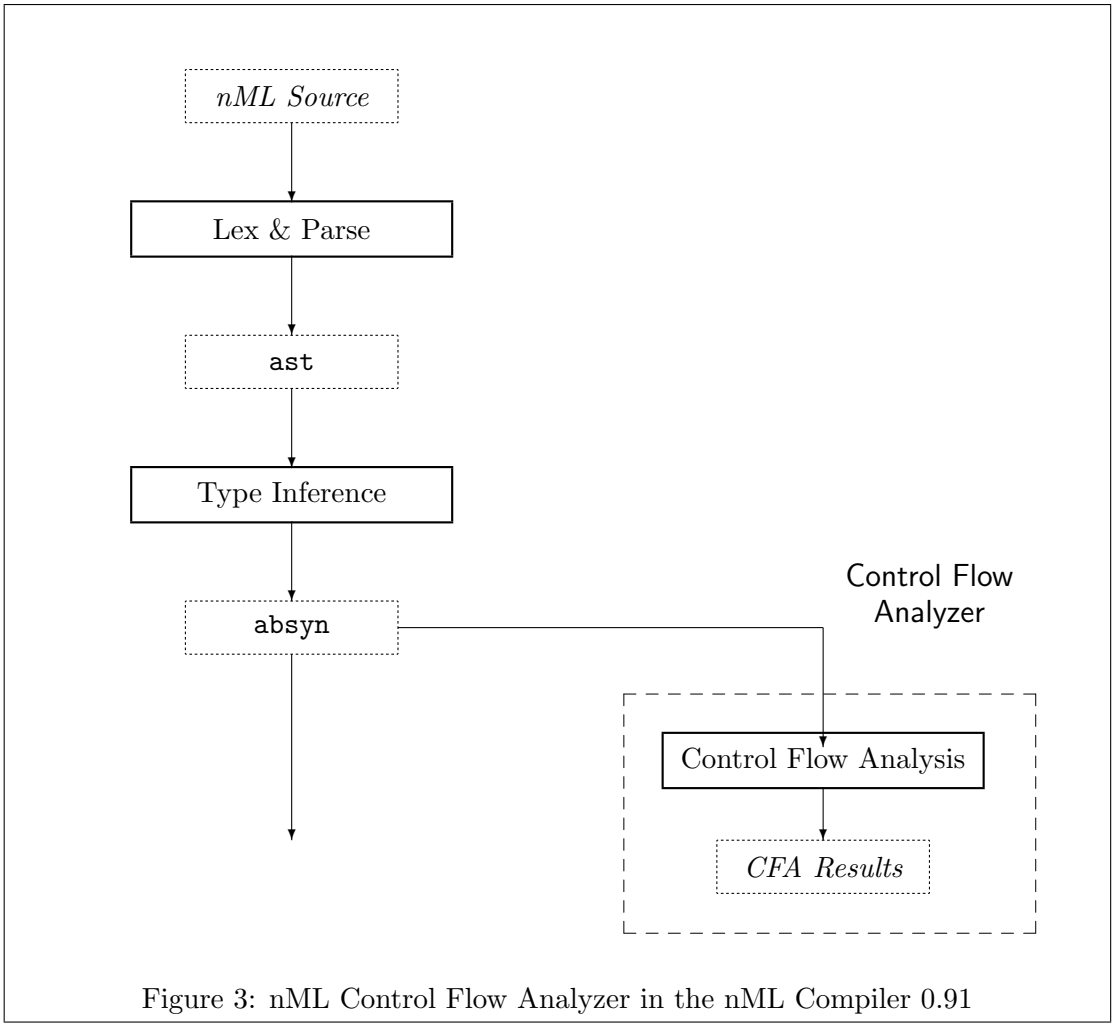


Figure 3: nML Control Flow Analyzer in the nML Compiler 0.91

list of the pre-analyzed files.

# 3 Availability

- You can get nML CFA by using cvs at `n/ncfa`.

# 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].