Elf Users Guide

Frank Pfenning

Draft of February 4, 1997 describing
Elf Version 0.5, August 30, 1995

Contents

1 Introduction

Currently, there exist three different Elf implementations. There is MLF (for Modular LF) which enriches the LF language with module-level constructs loosely based on ML's signatures and functors. Secondly, there is an implementation embedded into ML's top-level environment in which ML functions directly accomplish tasks such as loading files, executing queries, setting tracing flags, etc. Thirdly, there is an implementation of the core language as an Elf server which is intended to be executed as an inferior process to Emacs. It can also be executed in a shell (see Section8), but this is not generally recommended.

Elf should be understood as research software. This means comments, suggestions, and bug reports are extremely welcome, but, while I will do my best, I can make no guarantees regarding the response time. The same remark applies to to these notes which constitute the only "documentation" on the actual Elf implementations at present.

2 Lexical Conventions

The reserved characters in Elf are

```
%        one-line comment or pragma
%{ }%    delimited comment, braces must match
:        colon, type or kind declaration
.        period, end of signature entry
( )      parentheses, for grouping terms
[ ]      brackets, for lambda abstraction
{ }      braces, for pi quantification
```

All other characters can be included inside identifiers which are separated by whitespace or reserved characters. In particular, `A->B` is an identifier, whereas `A -> B` stands for the type of functions from `A` to `B`.

An uppercase identifier is one which begins with an underscore `_` or a letter in the range `A` through `Z`. A lowercase identifier begins with any other character except a reserved one. Numbers also count as lowercase identifiers and are not interpreted specially. Identifiers of the form `'id'` override the infix status of operators (see below). Free variables in a signature entry (declaration) must be uppercase, bound variables and constants may be either uppercase or lowercase identifiers. The identifiers `sigma`, `#pr#`, `` ` `` (backquote), and ``#`#`` are not treated special lexically, but predeclared (see below) and thus perhaps better not used in user programs at present.

Constants have static scope, which means that they can be shadowed by later subsequent declarations. A shadowed identifier (which can no longer be parsed) is printed as `%id%`.

3 Grammar

Here is the syntax for Elf in abbreviated form. Terms form the join of the three levels in the LF calculus, which are not distinguished syntactically. The comment approximates the meaning in LF.

```
sigentry ::= id : term.
query    ::= term.

term ::= type                % Type
       | term1 -> term2      % A -> B
       | term1 <- term2      % B -> A
       | {id : term1} term2  % \Pi x:A. K  or  \Pi x:A. B
       | [id : term1] term2  % \lambda x:A. B  or  \lambda x:A. M
       | term1 term2         % A M  or  M N
       | term : term         % explicit cast
       | _                   % hole, to be filled by term reconstruction
       | {id} term           % same as {id:_} term
       | [id] term           % same as [id:_] term
```

In the order of binding strength we disambiguate as follows:

1. Juxtaposition (application) is left-associative and binds the strongest.
2. `->` is right associative.
3. `<-` is left associative.
4. `:` is left associative.
5. {} and `[]` are weak prefix operators.

For example, the following are parsed identically:

```
d : a <- b <- {x} c x -> p x.
d : ({x} c x -> p x) -> b -> a.
d : ((a <- b) <- ({x:_} ((c x) -> (p x)))).
```

4 Fixity and Name Preference Declarations

Fixity declarations have the form

```
%infix <assoc> <prec> c1 ... cn
%prefix <prec> c1 ... cn
%postfix <prec> c1 ... cn
```

Here *assoc* is `left`, `right`, or `none`; *prec* is the precedence (the higher, the tighter) between 0 and 10000; *c1* through *cn* are the declared symbols. The declaration ends with the line. For example:

```
%infix left 10 + -
%infix left 20 * /
%prefix 30 ~

%infix right 5 &
%infix right 4 =>
%prefix 3 |-
```

Name preference declarations are given as

```
%name <fam> x1 ... xn
```

Here, *x1* through *xn* will be used as the names for variables whose type (family) is *fam*. Regular numbering will only be used when the list is exhausted. For example:

```
i : type.
%name i  x x' x''

o : type.
%name o  A B C

|- : o -> type.
%name |-  P
```

5 Term Reconstruction

The model of term reconstruction employed by Elf is straightforward, although it employs a relatively complex algorithm. "Term reconstruction" includes type reconstruction in the usual sense plus the reconstruction of implicit arguments to functions and filling in wildcards (written as an underscore _ in the input). The basic principle to keep in mind is the duality between omitted quantifiers and implicit arguments. For example,

```
o : type.
& : o -> o -> o.  %infix right 5 &
pf : o -> type.
andi : pf A -> pf B -> pf (A & B).
```

The last declaration will be printed back as one of the following two

```
andi : {A:o} {B:o} pf A -> pf B -> pf (A & B).
andi : {B:o} {A:o} pf A -> pf B -> pf (A & B).
```

which is the internal representation. Note that the quantifiers on `A` and `B` were omitted in the signature. This means that the corresponding arguments to `andi` remain implicit and will be reconstructed. For example,

```
([p:o] [u:pf p] andi u u) : {p:o} pf p -> pf (p & p).
```

is parsed into

```
([p:o] [u:pf p] andi _ _ u u) : {p:o} pf p -> pf (p & p).
```

and the two wildcards are determined to be `p`. Note that the second line is not a legal input, since the implicit arguments *must* be omitted. This is a necessary consequence of Elf's approach to term reconstruction, since the order of the implicit quantifiers (and therefore of the implicit arguments) is unspecified in the declaration of `andi` above.

Wildcards can also be explicitly given or they arise from implicit types in abstractions [x] or quantifiers x. For example,

```
([p] [u:pf p] andi u u) : {p} pf p -> pf (p & p).
```

is parsed into

```
([p:_] [u:pf p] andi _ _ u u) : {p:_} pf p -> pf (p & p).
```

and the values for the wildcards are determined as `o`, `p`, `p`, and `o`, respectively.

The algorithm for reconstruction employs an algorithm for solving constraints over dependently typed lambda-terms described in a paper "Unification and Anti-Unification in the Calculus of Constructions," LICS'91. Effectively, the value of an omitted argument may be determined from other arguments, or synthesized from the context in which an expression occurs. Here are some typical examples:

```
v : o -> o -> o.  %infix right 4 v
ori : pf A -> pf (A v B).
```

In an expression `(ori M)` which is parsed into `(ori _ _ M)`, the value of the first argument for, say, `A`, can be determined from `M`. However, the value of `B` can only be determined from the context it appears in. For example,

```
([p:o] [u:pf p] ori u) : {p:o} pf p -> pf (p v p).
```

determines both implicit arguments to `ori` as `p`. We refer to this situation by saying that in the declaration of `ori`, `A` can be synthesized, and both `A` and `B` can be inherited.

In some circumstances, an argument can neither be synthesized nor inherited. For example, consider the additional declarations

```
i : type.
all : (i -> o) -> o.
alle : pf (all A) -> pf (A T).
```

Now `T` can not always be inherited (and clearly not synthesized). For example,

```
([p:o] [u:pf (all [x:i] p)] alle u) : {p:o} pf (all [x:i] p) -> pf p.
```

but the implicit argument `T` to `alle` is inherently ambiguous (could be any term). Such a situation often leads to trouble during term reconstruction later on and is flagged with a warning by the front-end. In such a situation, it is desirable to make the argument explicit, as in

```
alle : {T:i} pf (all A) -> pf (A T).
```

Note that `A` can be synthesized and can therefore safely remain implicit.

In some circumstances it is useful to directly ascribe a type in order to disambiguate inherited arguments. For example

```
[p:o] [u:pf p] ori p
```

has type `{p:o} pf p -> pf (p v A p)` for any `A : o -> o`. We can disambiguate by ascribing a type to a subexpression. For example,

```
[p:o] [u:pf p] (ori p : pf (p v p))
```

In some circumstances, especially when arguments of some constants can be neither synthesized or inherited, a principal reconstruction for a given declaration does not exist even for expressions which have some type. In such a case the front-end issues an error message about remaining constraints after type reconstruction. The constraints are shown, but they are often difficult to interpret. As a general rule of thumb, (1) arguments which can be neither synthesized nor inherited should be made explicit, and (2) variables which occur only at the head of some application should be avoided. In the latter case the expression can often be disambiguated by explicitly ascribing types which mention the variable in question in a synthesizable or inheritable position.

When term reconstruction fails, the front-end issues an error message with the line number of the declaration in which the problem occurred and the disagreement encountered, printed in the form

```
<filename>:<location> Error:
Type checking failed on declaration of c
M : A <> B
Unification failure due to <reason>
```

which means that the type inferred for `M` was `A` but that some constraints required it to be equal to `B`, and `A` and `B` are different (that is, do not unify). The reason why `A` and `B` do not unify is given in the next line. The *filename* and *location* information can be used by Elf's Emacs interface to jump to the specified location in the given file for editing of the incorrect declaration for the constant `c`. The *location* has the form `line1.column1-line2.column2` and represent Elf's best guess as to the source of the error. Due to the propagation of non-trivial constraints the source of a type reconstruction failure can sometimes not be pinpointed. In that case the error message has the form

```
<filename>:<location> Error:
Type checking failed on declaration of c
M : A
is inconsistent with other constraints
Unification failure due to <reason>
```

When the reason for failure of type reconstruction is elusive, a recommended strategy is to explicitly ascribe types to subterms occurring in the declaration, using the form `(M : A)`. It is also possible to trace type reconstruction; this feature will be made available in the Elf server at some future time.

Normally, error and trace messages are printed in reparsable format (see Section6), but sometimes this is impossible—the printer then falls back on internal form and surrounds every externally unprintable term in double quotes.

6 Printing

In accordance with the description above, there are two ways of printing terms. One is a reparsable format, in which infix identifers are printed in infix and implicit arguments are omitted. The other shows the internal form in order to allow the user to inspect the result of type reconstruction. Normally, signatures are printed back in internal form after reconstruction, and answer substitutions after a successful solution to a query are printed in reparsable form. Constants `c` which have been shadowed (and can therefore no longer be parsed) are printed as `%c%`. A signature file can be printed with Elf server commands

```
print_sig <filename>
```

to see the reparsable version, and

```
print_sig_full <filename>
```

to see all implicit arguments.

Different sorts of variables are printed differently at the moment to aid in their recognition. Existential variables (or logic variables, in the logic programming terminology) are simply printed as free variables, the way they would occur in input. Universal variables (or parameters, in the terminology of logical frameworks) are printed as `!u`, and a free variable in a clause is printed as `^F`. Note that free variables in a clause are interpreted *universally*, just as in ML. They may not be instantiated by type reconstruction!

There are also a few variables controlling the appearance of the terms as they are printed shown below.

- `limit printDepth` n, `unlimit printDepth`

- limit printLength n, unlimit printLength
- limit iprintDepth n, unlimit iprintDepth
- limit iprintLength n, unlimit iprintLength

Here printDepth and printLength refer to the printing in external, reparsable form, while iprintDepth and iprintLength refer to the internal, fully explicit form.

7 Special Families

There are two type families which are given a special interpretation during execution of a program: sigma and ‘ (backquote). Backquote was intended as a debugging aid, but the semantics is to tricky to make this reliable in the current intepreter. The first is generally useful at the top-level.
sigma. Use ?- sigma [x:A] B. to get the operational behavior of solving A with proof term M, then solving [M/x]B. The corresponding deduction is a pair of the witness M and deduction of [M/x]B, using the proof constructor #pr#. This awkward mechanism will be replaced by the solve declaration in the module system. Sigma's can be nested, but they can only be used at the top-level and cannot be embedded in signatures, since they are not properly part of the type theory.
‘ (backquote). Use ‘ A for tracing invocations of the goal A. Note that this will break proof transformation code, since the type of the constants now involves backquote. The constructor #‘# coerces an object of type A into an object of type ‘ A and will thus appear in proof objects if a subgoal of the form ‘ A was successful.

8 Elf Server Shell Command Summary

Below is the summary of the commands which can be given directly to the Elf server in a shell. This is not the recommended mode of interaction, since this interface was designed for an inferior process to Emacs. Note that *configuration* refers to the *contents* of the CONFIG file, not its name.

```
configure <configuration>  --- configure Elf server
check-config               --- check current configuration
check-file <filename>      --- check only given <filename>
solve <Return> <query>.    --- find first solution to <query>
solve* <n> <Return> <query>.  --- find first <n> solutions to <query>
top                        --- start Prolog-like top level
cd <directory>             --- change working directory
pwd                        --- print working directory
quit                       --- quit server

toc                        --- show files comprising current configuration
show_prog                  --- show names of declared constants
print_sig <filename>       --- print signature in external form
print_sig_full <filename>  --- print signature in internal form
type-of <constant>         --- print type of constant in external form

chatter <n>                --- set level of diagnostic chatter (default 2)
trace <n>                  --- set trace level for query execution (default 0)
```

```
limit <param> <n>           --- limit printing parameter
unlimit <param>             --- unlimit printing parameter
```

9 Elf Server Emacs Command Summary

The Emacs interface to Elf provides support for

- editing Elf source files with reasonable indentation
- managing configurations of Elf source files, including TAGS tables
- communication with an inferior Elf server to type-check and execute declarations and queries
- interaction with an inferior ElfSML process
- pop-up and pull-down menus for common operations (see `elf-menus.el`)
- syntax highlighting for Elf code (see `elf-hilit.el` for Hilit19 or `elf-font.el` for font-lock). These depend on lower-level features of Hilit19 and the Font Lock package and are quite unstable. You must currently load those explicitly with `M-x load-library elf-font` or `M-x load-library elf-hilit`.

We recommend the use of XEmacs, but most features (with the exception of font-lock mode) should also work in FSF Gnu-Emacs, Versions 19.xx. The Emacs code is relatively well documented, and you can use standard help functions such as `C-h m` (to get help on the current major mode) or `C-h f <function>` (to get help on an individual function), or simply browse through the sources[1].

To use the Elf mode effectively, put into your `.emacs` file:

```
(setq load-path (cons "/afs/cs/user/fp/courses/comp-ded/lib/emacs" load-path))

(autoload 'elf-mode "elf-menus" "Major mode for editing Elf source." t)
(autoload 'elf-server "elf-menus" "Run an inferior Elf server." t)
(autoload 'elfsml "elf-menus" "Run an inferior ElfSML process." t)

(setq auto-mode-alist
      (cons '("\\.elf$" . elf-mode)
            (cons '("\\.quy$" . elf-mode)
                  auto-mode-alist)))

(setq elf-server-program "/afs/cs/user/fp/courses/comp-ded/bin/elf-server")
(setq elfsml-program "/afs/cs/user/fp/courses/comp-ded/bin/elfsml")
```

The documentation below was generated with `C-h m` while in Elf mode in Emacs. My apologies for the long verbatim environment, but it is difficult to keep this information updated and consistent.

[1] file:/afs/cs/user/fp/courses/comp-ded/lib/emacs/

```
Elf Mode:
Major mode for editing Elf code.
Tab indents for Elf code.
Delete converts tabs to spaces as it moves back.
M-C-q indents all lines in current Elf paragraph (declaration or query).

Elf mode also provides commands to maintain groups of Elf source files
(configurations) and communicate with an Elf server which type-checks or
executes declarations or queries.  It also supports quick jumps to the
(presumed) source of error message that may arise during parsing or
type-checking.

Customisation: Entry to this mode runs the hooks on elf-mode-hook.
See also the hints for the .emacs file given below.

Mode map
========

TAB             elf-indent-line
DEL             backward-delete-char-untabify
button3         elf-menu
C-c             Prefix Command
C-i             elf-indent-line
M-C-q           elf-indent-paragraph

C-c TAB         elf-server-interrupt
C-c LFD         elf-server-send-newline
C-c .           elf-complete
C-c ;           elf-server-send-semicolon
C-c =           elf-goto-error
C-c ?           elf-completions-at-point
C-c `           elf-next-error
C-c c           elf-type-const
C-c e           elf-expected-type-at-point
C-c p           elf-type-at-point
C-c q           tags-query-replace
C-c s           tags-search
C-c C-c         elf-save-check-config
C-c C-d         elf-check-declaration
C-c C-e         elf-solve-query
C-c C-i         elf-server-interrupt
C-c C-j         elf-server-send-newline
C-c C-q         elf-check-query
C-c C-s         elf-save-check-file


Overview
```

```
========

The basic architecture is that Emacs sends commands to an Elf server which
runs as an inferior process, usually in the buffer *elf-server*.  Emacs in
turn interprets or displays the replies from the Elf server.  Since a typical
Elf application comprises several files, Emacs maintains a configuration
in a file, usally called CONFIG.  This file contains declarations of the form
  -static FILENAME
  -dynamic FILENAME
  -query FILENAME
for static, dynamic, and query files, respectively.  The declarations
must be in dependency order.  A configuration is established with the
command M-x elf-server-configure.

When a new file is switched to Elf mode (typically done automatically if a
file has extension `.elf' or `.quy' and the `auto-mode-alist' is set correctly
(see below)), the user is asked whether to add the new file to the current
configuration.  The CONFIG file will be updated automatically if necessary.

The files in the current configuration can be checked in sequence with
C-c C-c, queries can be sent with C-c C-e.  An optional prefix argument
specifies how many solutions to search for, 0 means ask interactively, and -1
to find all.  If a type error should arise during these or related operations,
the command C-c ` visits the presumed source of the type error in a separate
buffer.

Summary of most common commands:
 M-x elf-server                        start Elf server
 M-x elf-server-configure              configure Elf server from CONFIG file
 M-x elf-save-check-config C-c C-c  save, check & load configuration
 M-x elf-save-check-file   C-c C-s  save, check & load current file
 M-x elf-check-declaration C-c C-d  type-check declaration at point
 M-x elf-solve-query       C-c C-e  execute query at point
 M-x elf-server-display    M-x elf-server-display  display Elf server buffer
 M-x elf-tag                           create TAGS file of current configuration

There are a number of commands to check declarations or obtain type
information from the Elf server.  However, the only commands that actually
change the global signature consulted by the type checker are the file-level
commands `elf-check-config', `elf-save-check-config'(C-c C-c), and
`elf-save-check-file'(C-c C-s).  Thus, it will occasionally be necessary to
call one of these commands (typically C-c C-s) to make new declarations
available to subsequent declarations in the same file.  Since the Elf syntax
does not distinguish between declarations and queries, Emacs assumes the
current paragraph is a declaration unless the Query minor mode has been
switched on (see `elf-query-mode').
```

```
Individual Commands
===================

Configurations, Declarations, and Queries

  elf-save-check-config                   C-c C-c
   Save its modified buffers and then check the current Elf configuration.
   With prefix argument also displays Elf server buffer.
   If necessary, this will start up an Elf server process.

  elf-save-check-file                     C-c C-s
   Save buffer and then check it by giving a command to the Elf server.
   With prefix argument also displays Elf server buffer.

  elf-check-declaration                   C-c C-d
   Send the current declaration to the Elf server process for checking.
   With prefix argument, subsequently display Elf server buffer.

  elf-check-query                         C-c C-q
   Send the current query to the Elf server process for type-checking.
   Note that this will not execute the query (see function elf-solve-query).
   With prefix argument also display Elf server buffer.

  elf-solve-query                         C-c C-e
   Send the current query to the Elf server process to be solved.
   Variable `elf-solve-default' determines how many solutions to search for
   (default: 1).  An optional prefix argument overrides this variable: 0 means
   to wait after each solution (as in Prolog's top level), -1 means to find
   all solutions.  You can ask for more solutions with C-c ;
   or return to the server's command interpreter with C-c LFD.

Subterm at Point

  elf-type-at-point                       C-c p
   Display the type of the subterm at point in the current Elf paragraph.

   The subterm at point is the smallest subterm whose printed representation
   begins to the left of point and extends up to or beyond point.  After this
   and similar commands applicable to subterms, the current region (between
   mark and point) is set to encompass precisely the selected subterm.  In
   XEmacs, it will thus be highlighted under many circumstances.  In other
   versions of Emacs C-x C-x will indicate the extent of
   the region.

   The type computed for the subterm at point takes contextual information
   into account.  For example, if the subterm at point is a constant with
   implicit arguments, the type displayed will be the instance of the constant
```

```
  (unlike M-x elf-type-const (C-c c), which yields the absolute
  type of a constant).

 elf-expected-type-at-point              C-c e
  Display the type expected at the point in the current declaration.

  This replaces the subterm at point by an underscore _ and determines the
  type that _ would have to have for the whole declaration to be valid.  This
  is useful for debugging in places where inconsistent type constraints have
  arisen.  Error messages may be given, but will not be correctly interpreted
  by Emacs, since the string sent to the server may be different from the
  declaration in the buffer.

 elf-type-const                          C-c c
  Display the type of the constant before point.
  Note that the type of the constant will be 'absolute' rather than the
  type of the particular instance of the constant.

 elf-completions-at-point                C-c ?
  List the possible completions of the term at point based on type
  information.

  The possible completions are numbered, and the function elf-complete
  (C-c .) can be used subsequently to replace the term at point
  with one of the alternatives.

  Above the display of the alternatives, the type of the subterm at
  point is shown, since it is this type which is the basis for listing
  the possible completions.

  In the list of alternatives, a variable X free in the remaining declaration
  is printed ^X, and a bound variable x may be printed as !x.  These marks
  are intended to aid in the understanding of the alternatives, but
  must be removed in case the alternative is copied literally into the
  input declaration (as, for example, after the C-c . command).

 elf-complete                            C-c .
  Pick the alternative N from among possible completions.
  This replaces the current region with the given pattern.  The list of
  completions must be generated with the command elf-completions-at-point
  (C-c ?).

Server State

 elf-server                              M-x elf-server
  Start an Elf server process in a buffer named *elf-server*.
```

```
  elf-server-interrupt                    C-c TAB
   Interrupt the Elf server-process.

  elf-server-quit                         M-x elf-server-quit
   Kill the Elf server process.

  elf-server-restart                      M-x elf-server-restart
   Restarts server and re-initializes configuration.
   This is primarily useful during debugging of the Elf server code or
   if the Elf server is hopelessly wedged.

Tags (for other, call C-h tag or see ‘etags’ documentation)

  elf-tag                                 M-x elf-tag
   Create tags file TAGS for current configuration CONFIG.
   If current configuration is names CONFIGx, tags file will be named TAGx.
   Errors are displayed in the Elf server buffer.

Error Handling

  elf-next-error                          C-c ‘
   Find the next error by parsing the Elf server or ElfSML buffer.

  elf-goto-error                          C-c =
   Go to the error reported on the current line or below.

Server Parameters

  elf-chatter                             M-x elf-chatter
   Sets the level of chatter in the Elf server.  Default is 2.

  elf-trace                               M-x elf-trace
   Sets the level of tracing in the Elf server.  Default is 0 (no tracing).

Editing

  elf-indent-region                       M-x elf-indent-region
   Indent each line of the region as Elf code.

Minor Modes
===========

Associated minor modes are Query (usually switched on automatically
in query files, explicitly toggled with elf-query-mode) and 2ElfSML
(toggled with elf-to-elfsml-mode).

Related Major Modes
```

```
===================

Related major modes are Elf Server (for the Elf server buffer) and ElfSML (for
an inferior ElfSML process).  Both modes are based on the standard Emacs
comint package and inherit keybindings for retrieving preceding input.

Customization
=============

The following variables may be of general utility.

 elf-solve-default default number of solutions searched for
                   in C-c C-e; 0 means interactive, -1 means all.
                   Override with a prefix argument to C-c C-e.

 elf-indent        amount of indentation for nested Elf expressions

 elf-infix-regexp  matches infix operators that are treated special for
                   purposes of indentation

 elf-save-silently if non-nil, modified buffers in the current configuration
                   will be saved without confirmation during C-c C-c

 elf-server-program  full pathname of Elf server program
 elfsml-program      full pathname of ElfSML program

The following is a typical section of a .emacs initialization file.

; If elf.el lives in some non-standard directory, you must tell emacs
; where to get it. This may or may not be necessary.
(setq load-path (cons "/afs/cs/project/ergo/research/elf/emacs" load-path))

(autoload 'elf-mode "elf" "Major mode for editing Elf source." t)
(autoload 'elf-server "elf" "Run an inferior Elf server." t)
(autoload 'elfsml "elf" "Run an inferior ElfSML process." t)

(setq auto-mode-alist
      (cons '("\\.elf$" . elf-mode)
            (cons '("\\.quy$" . elf-mode)
                  auto-mode-alist)))

; The Elf server binary has an odd name or location
(setq elf-server-program "/afs/cs/project/ergo/research/elf/bin/elf-server")
(setq elfsml-program "/afs/cs/project/ergo/research/elf/bin/elfsml")

; Sample customization of key bindings for Elf mode
;(setq elf-mode-hook '((lambda ()
```

```
; (define-key elf-mode-map "\M-\t" 'comint-replace-by-expanded-filename)
; )))
```

10 A Typical Session

We assume you are in Emacs, visiting an Elf source file in Elf mode. The most commonly used commands when you browse some files are:

```
M-x elf-server-configure   ; select default CONFIG
C-c C-c                    ; type-check and load current configuration
C-x C-f examples.quy       ; visit example query file
C-c C-e                    ; execute current query
```

If you are developing your own code, you should set up your own `CONFIG` file and add to it as you go along. For writing Elf code, you should consider the syntax highlighting which you can enabe from the Menu bar in XEmacs (but at present not in FSF Gnu Emacs). In addition to the commands above, you will likely need the following.

```
Tab                        ; smart indent line
C-c C-s                    ; type-check and load current file only
C-c C-d                    ; type-check current declaration only
C-c `                      ; goto next error location in source
C-c C-i                    ; interrrupt Elf server
```

11 Common Problems

Dynamic vs. Static Families. A common source of problems is that dynamic families are loaded/interpreted statically, or static families are loaded/interpreted dynamically. Try to keep this distinction straight: each file should contain only static or dynamic declarations. Common symptoms:

1. the query succeeds when it should fail, because some families which intuitively should be treated subgoals are static. Remember that any query `A` will succeed if `A` is static!

2. the query does not terminate, because some families which should be treated as static are dynamic. For example, if the declarations

   ```
   nat : type.
   s : nat -> nat.
   z : nat.
   ```

   are loaded dynamically, then we may get non-terminating behavior in circumstances where a free variable of type nat should remain in the answer. For example,

   ```
   eq : nat -> nat -> type.
   id : eq N N.
   ?- eq N N.
   ```

does not terminate if both `eq` and nat `are` dynamic. One should also keep in mind that it does not make sense for a static family to depend on a dynamic one. For example, if the query has the form `a M1 ... Mn` and `a` is a static type family then no free variable in `M1 ... Mn` will be solved, regardless of whether it is static or dynamic.

Whitespace and Identifiers. Unlike ML, even symbolic identifiers must be surrounded by whitespace. For example, `a->b` is one identifier. The only exceptions are the characters `.:()[]{}` which terminate identifiers.

Wildcards. Wild cards _ are existentially quantified when a clause is read, free variables (undeclared identifiers whose first letter is upper-case) are universally quantified! Wild cards should be used sparingly, and only if the omitted argument is uniquely determined by the context.

Static Scoping. After reloading an intermediate file, other files which depend on it must also be reloaded. Currently, if the checking of a file fails, none of the declarations in the file are entered into the top-level signature. I realize this is annoying on occasion, since we cannot re-check individual declarations unless the file has loaded successfully at least once. This particular feature might change in future versions.

12 Known Bugs

- Currently, there is no acyclicity check for universes in the type inference engine.

  Consequence: The system behaves as if `type:type`. This does not matter for the applications, but after some hard work one could obtain non-normalizing terms and wreak havoc on the system.

  Workaround: Don't try to take advantage of the fact that the implementation considers `type:type`.