fl — A Functional Language for Formal Verification User's Guide

Carl-Johan H. Seger
Email: cjhseger@gmail.com

October 16, 2018

Abstract

Fl consists essentially of five main parts: a general, strongly typed, functional language, an efficient implementation of Ordered Binary Decision Diagrams (OBDDs) built into the functional language, an efficient SAT solver integrated into the language, extensive visualization capabilities, and a symbolic simulation engine for Verilog RTL designs. Since the interface language to fl is a fully general functional language in which OBDDs and an efficient SAT solver have been built in, the verification system is not only useful for carrying out symbolic trajectory evaluation (STE), but also for experimenting with various verification (formal and informal) techniques that require the use of OBDDs and/or SAT solvers.

This document is intended as both a user's guide and (to some extent) a reference guide.

Contents

1 Fl—The Meta Language of VossII

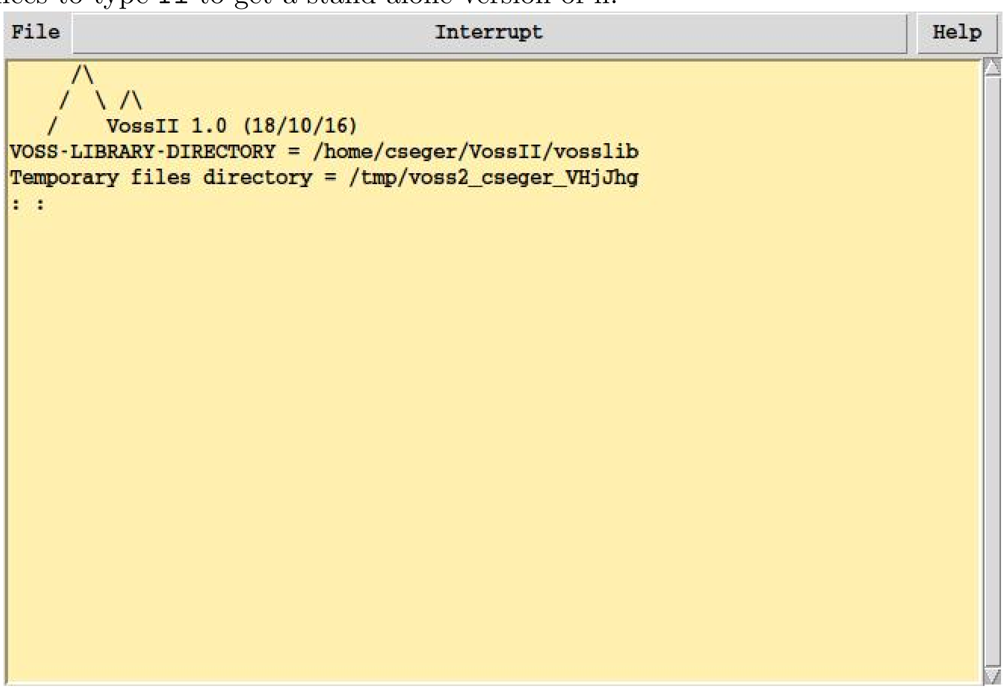
In this section we provide an introduction to the functional language fl.

Similar to many theorem provers (e.g., the HOL system [25, 26]) the VossII command language for the verification system is a general purpose programming language. In fact, the fl language shows a strong degree of influence from the version of ML used in the HOL-88 system. However, there are several differences: many syntactic but some more fundamental. In particular, the functional language used in VossII has lazy evaluation semantics. In other words, no expression is evaluated until it is absolutely needed. Similarly, no expression is evaluated more than once. Another difference is that Boolean functions are first-class objects and can be created, evaluated, compared and printed out. For efficiency reasons these Boolean functions are represented as ordered binary decision diagrams.

Fl is an interactive language. At top-level one can for example: define functions (possibly of arity 0), define new concrete types, evaluate expressions, and modify the parser. In this section we introduce the language by several examples.

1.1 Invoking fl

If the VossII system is installed on your system and you have the suitable search path set up, it suffices to type `fl` to get a stand-alone version of fl.



Note that the VOSS-LIBRARY-DIRECTORY is installation dependent. We return to this below and in Section 5.

The fl program can take a number of arguments. In particular,

-f n Start fl by first reading in the content of the file named n.

-I dir Set the default search directory to dir.

-noX or noX do not use the graphical (X-windows) interface. Useful when running batch oriented jobs. Note that any calls to graphics primitives, will fail with a run time exception when fl is run in the -noX mode.

-use_stdin or use_stdin Read inputs also from stdin as well as from the graphical interface.

-use_stdout use_stdout Read inputs also from stdin as well as from the graphical interface.

read_input_from_file filename Read inputs continously from the file 'filename'.

write_output_to_file filename Write outputs to the file 'filename' in addition to the graphical user interface.

-r i Initialize the random number generator with the seed i. This allows the `rvariable` command to create new sets of random variable values. See the rvariable command description in Section 3.1 for more details.

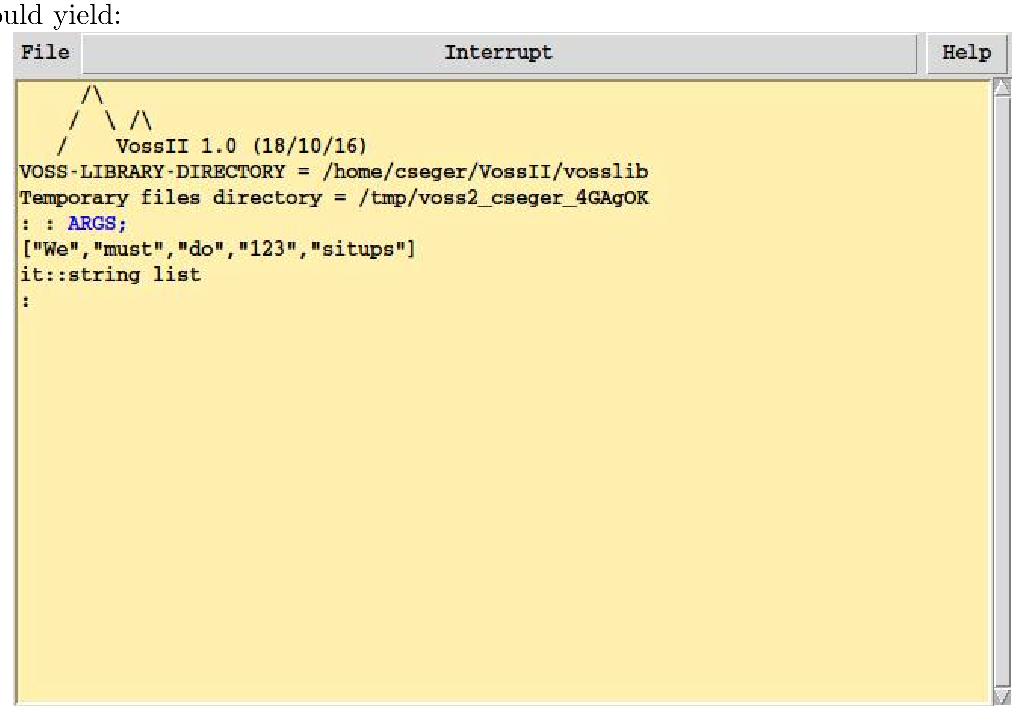
-v fn Store the variable ordering obtained by dynamic variable re-ordering in the file fn.

-h or help Print out the available options and quit.

Any additional arguments to fl will be stored in the fl expression ARGS as a list of strings. For example:

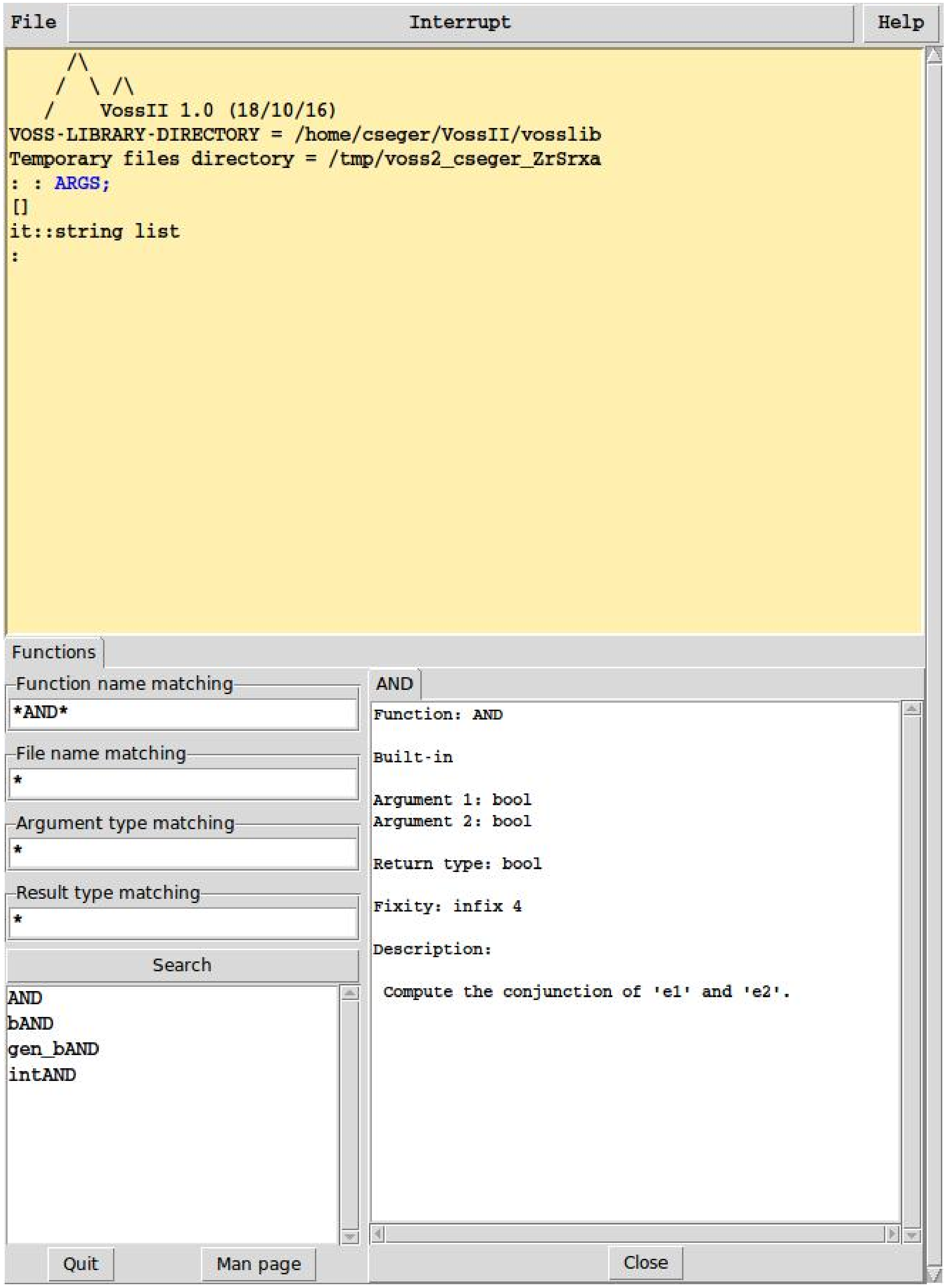
```
% fl We must do 123 situps
```

would yield:



1.2 Help system

To make it easier to use fl, there is an automatically generated help system available by pushing the "Help" button in top rigth corner of the user interface. Whenever a function is defined, it is added to the online help system. Furthermore, if the function declaration is preceeded by some comments (lines started with //), the comments will be displayed together with information of the fixity (if any), number and types of input arguments, as well as the type of the result of the function. The help system allow the user to serach by name, file, argument type(s) or resulting type using regular globbing style patterns. Also, if the function is defined by fl code, there will be a live link to the code used to define the function. For example, searching for functions with the word AND in them yield:



1.3 Expressions

The Fl prompt is : so lines beginning with this contain the user's input; all other lines are output of the system.

```
File                 Interrupt                 Help
: 2+3;
5
it::int
:
```

Here we simply evaluated the expression 2+3 and fl reduced it to normal form; in this case computed the result 5.

Note that VossII stores the result of the most recent expression in a variable called `it`. Thus, continuing the example by evaluating the expression `it;` yields:

```
File                 Interrupt                 Help
: it;
5
it::int
:
```

1.4 Declarations

The declaration let x = e binds a computation of e to the variable x. Note that it does not evaluate e (since the language is lazy). Only if x is printed or used in some other expression that is evaluated will it be evaluated. Also, once e is evaluated, x will refer to the result of the evaluation rather than the computation. Hence, the expression e is evaluated at most once, but it may not be evaluated at all.

```
File                 Interrupt                 Help
: let x = 3+3;
x::int
:
```

Note that when expressions are bound to variables, the system simply prints out the inferred type of the expression. We will return to the typing scheme in fl later. For now, it suffices to say that fl tries to find as general type as possible that is consistent with the type of the expression.

A declaration can be made local to the evaluation of an expression e by evaluating the expression decl in e. For example:

```
File                 Interrupt                 Help
: let y = let x = 4 in x-5;
y::int
:
```

would bind the expression 4 to x only inside the expression bound to y. Thus, we get:

```
File                    Interrupt                    Help
: let x = 2;
x::int
: let y = let x = 4 in x-5;
y::int
: x;
2
it::int
: y;
-1
it::int
:
```

fl is lexically scoped, and thus the binding in effect at the time of definition is the one used. In other words, if we write:

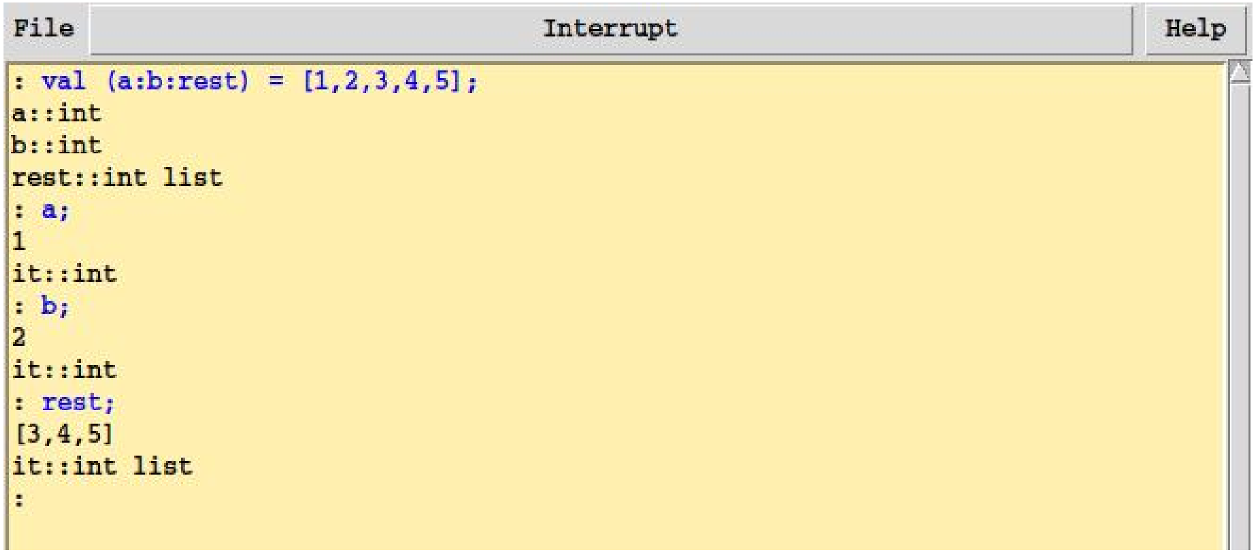
```
File                    Interrupt                    Help
: let x = 2;
x::int
: let y = x*5;
y::int
: let x = 12;
x::int
:
```

and we then evaluate y we will get 10 rather than 60.

To bind several expressions to several variables at the same time, a special keyword `val` is available to take a complicated object apart automatically. For example, if q is an expression of type (int#bool) then we could write:

```
File                    Interrupt                    Help
: let top_level q =
        val (i,b) = q in
        i < 3 => b | F
;
top_level::(int#bool)->bool
: top_level (12,variable "a");
F
it::bool
: top_level (-2,variable "a");
a
it::bool
:
```

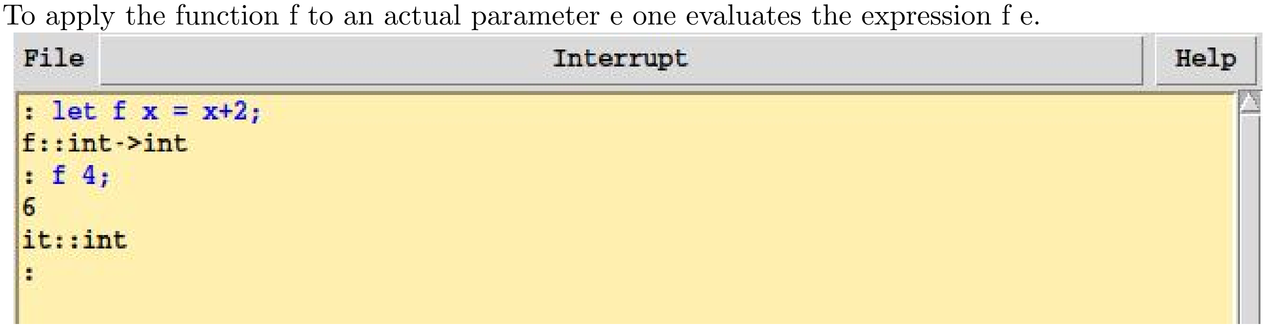
or



In general, the expression to the right of the val keyword can be an arbitrary complex pattern similar to the patterns allowed in function definitions and lambda expressions. For more details, see the section on pattern matching on page **??**.

1.5 Functions

To define a function f with formal parameter x and body e one performs the declaration: let f x = e. To apply the function f to an actual parameter e one evaluates the expression f e.



Note that the type inferred for f is essentially "a function taking an int as argument and returning an int". Applications binds more tightly than anything else in fl; thus for example: f 3 * 4 would be evaluated as: ((f 3)*4) and thus yield 20.

Functions of several arguments can also be defined:

```
File                         Interrupt                         Help
: let add x y = x+2*y;
add::int->int->int
: add 1 4;
9
it::int
: let f = add 1;
f::int->int
: f 4;
9
it::int
:
```

Applications associate to the left so add 3 4 means (add 3) 4. In the expression add 3, the function add is partially applied to 3; the resulting value is the function of type int→int which adds 3 to twice its argument. Thus add takes its arguments one at a time. We could have made add take a single argument of the cartesian product type (int#int):

```
File                         Interrupt                         Help
: let add (x,y) = x+2*y;
add::(int#int)->int
: add (3,4);
11
it::int
:
```

As well as taking structured arguments (e.g. (3,4)) functions may also return structured results:

```
File                         Interrupt                         Help
: let manhat_dist (x1,y1) (x2,y2) = (x2-x1, y2-y1);
manhat_dist::[-,-] (*#**)->(***#****)->(*****#******)
: manhat_dist (1,1) (3,5);
(2,4)
it::int#int
: let geometric_dist (x1,y1) (x2,y2) = sqrt ( (pow 2.0 (x1-x2)) + (pow 2.0 (y1-y
2)));
geometric_dist::(float#float)->(float#float)->float
: geometric_dist (1.0,1.0) (3.0,5.0);
0.559017
it::float
:
```

The latter function illustrates the use of floats as well as integers.

Trying to print a function with insufficient number of actual arguments yield a dash for the function and the type of the expression is printed out. For example:

```
: (5, manhat_dist (1,2));
(5,,-)
it::int#(int#int)->(int#int)
:
```

The only exception to this rule is for concrete types for which the user has installed a printing function. For more details of concrete types, see page **??**.

1.6 Recursion

The following is an attempt to define the factorial function:

```
: let fact n = n=0 => 1 | n*fact (n-1);
===Type error around line 4
Unidentified identifier "fact"
```

The problem is that any free variables in the body of a function have the bindings they had just before the function was declared; fact is such a free variable in the body of the declaration above, and since it is not defined before its own declaration, an error results. To make things clear consider:

```
: let f n = n+1;
f::int->int
: let f n = n=0 => 1 | n*f (n-1);
f::int->int
: f 3;
9
it::int
:
```

Here 3 results in the evaluation of 3*(f 2), but now the first f is used so f 2 evaluates to 2+1=3. To make a function declaration hold within its own body, letrec instead of let must be used. The correct recursive definition of the factorial function is thus:

```
: letrec fact n = n=0 => 1 | n*fact (n-1);
fact::int->int
: fact 5;
120
it::int
:
```

It should be pointed out that fl currently does not allow direct definition of mutually recursive functions. For an example on how this limitation can be dealt with, see the subsection on concrete type decalartions.

1.7 Tuples

If $e\_1, e\_2, \ldots, e\_n$ have types $t\_1, t\_2, \ldots, t\_n$, then the fl expression $(e\_1, e\_2, \ldots, e\_n)$ have type t_1#t_2#...#t_n. The standard functions on tuples are fst (first), snd (second), and the infix operation , (pair).

```
File                         Interrupt                         Help
: let q = ((1,2),3);
q::(int#int)#int
: let qq = (1,2,3);
qq::int#int#int
: q;
((1,2),3)
it::(int#int)#int
: qq;
(1,2,3)
it::int#int#int
: let qqq = (1,"abc");
qqq::int#string
: qqq;
(1,"abc")
it::int#string
:
```

1.8 Lists

If $e\_1, e\_2, \ldots, e\_n$ have type t, then the fl expression $[e\_1, e\_2, \ldots, e\_n]$ has type (t list). The standard functions on lists are hd (head), tl (tail), [] (the empty list), and the infix operation : (cons). Note that all elements of a list must have the same type (compare this with a tuple where the size is determined but each member of the tuple can have different type).

```
File                                Interrupt                                Help
: let l = [1,2,3,3,2,1,2];
l::int list
: hd l;
1
it::int
: tl l;
[2,3,3,2,1,2]
it::int list
: 0:l;
[0,1,2,3,3,2,1,2]
it::int list
: length l;
7
it::int
: letrec odd_even (a:b:rem) =
    val (r_odd, r_even) = odd_even rem then
    (a:r_odd), (b:r_even)
 /\    odd_even [a] = [a],[]
 /\    odd_even [] = [], []
;
odd_even::(* list)->((* list)#(* list))
: val (odd,even) = odd_even (1--20);
odd::int list
even::int list
: odd;
[1,3,5,7,9,11,13,15,17,19]
it::int list
: even;
[2,4,6,8,10,12,14,16,18,20]
it::int list
:
```

There are a large number of list functions built into fl. For example:

File Interrupt Help

```
: let l1 = 1 upto 8;
l1::int list
: let l2 = 13 downto 1;
l2::int list
: let l3 = 1--100;
l3::int list
: l1 @ l2;
[1,2,3,4,5,6,7,8,13,12,11,10,9,8,7,6,5,4,3,2,1]
it::int list
: lastn 5 l1;
[4,5,6,7,8]
it::int list
: butlast l2;
[13,12,11,10,9,8,7,6,5,4,3,2]
it::int list
: firstn 7 l3;
[1,2,3,4,5,6,7]
it::int list
: butfirstn 96 l3;
[97,98,99,100]
it::int list
: cluster 4 (1--20);
[[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16],[17,18,19,20]]
it::(int list) list
: let ll = [l1, l2, [4,8,12]];
ll::(int list) list
: flat ll;
[1,2,3,4,5,6,7,8,13,12,11,10,9,8,7,6,5,4,3,2,1,4,8,12]
it::int list
:
```

To find all such functions, use the help system and search for functions whose argument type matches "*list*".

1.9 Strings

A sequence of characters enclosed between " is a string. The standard functions on strings are ^ (catenation), explode (make string into list of strings) and implode (make list of strings into single string). There a numerous other string functions. For example:

```
File                         Interrupt                         Help
: let q = "abc and _12!@@#";
q::string
: let qq = "qw\"q qw";
qq::string
: q^qq;
"abc and _12!@@#qw"q qw"
it::string
: explode q;
["a","b","c"," ","a","n","d"," ","_","1","2","!","@","@","#"]
it::string list
: implode ["1", "2"];
"12"
it::string
: filter (str_is_prefix "a") ["abc", "Ava", "vad", "bbAb", "aaa"];
["abc","aaa"]
it::string list
: filter (str_is_suffix "a") ["abc", "Ava", "vad", "bbAb", "aaa"];
["Ava","aaa"]
it::string list
: filter (str_match "*a*") ["abc", "Ava", "vad", "bbAb", "aaa"];
["abc","Ava","vad","aaa"]
it::string list
: qsort strcmp ["abc", "Ava", "vad", "bbAb", "aaa"];
["Ava","aaa","abc","bbAb","vad"]
it::string list
: let s = "A funny story often told is the story of";
s::string
: strlen s;
40
it::int
: strstr s "is";
26
it::int
: substr s 1 25;
"A funny story often told "
it::string
: substr s 26 (-1);
"is the story of"
it::string
: sprintf "%s: %032b (%d)" "abc" 1423 1423;
"abc: 00000000000000000000010110001111 (1423)"
it::string
:
```

1.10 Polymorphism

The list processing functions hd, tl, etc. can be used on all types of lists.

File | Interrupt | Help

```
: hd [1,2,3];
1
it::int
: hd ["abc", "edf"];
"abc"
it::string
: (hd ["a", "b"]), hd [4,2,1];
("a",4)
it::string#int
: let q = [T,T,F];
q::bool list
: hd q;
T
it::bool
:
```

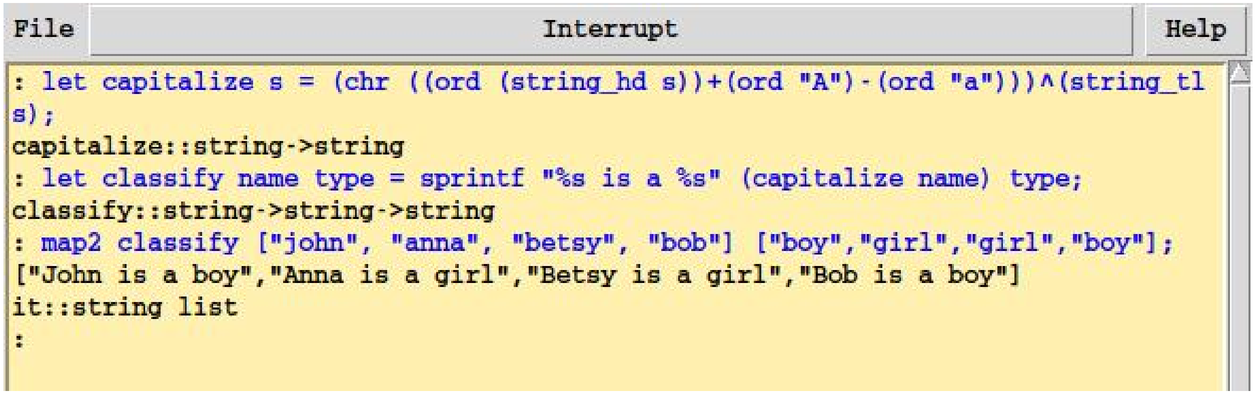
Thus hd has several types; for example, it is used above with types (int list) →int, (string list) →string, and (bool list) →bool. In fact if ty is any type then hd has the type (ty list) →ty. Functions, like hd, with many types are called polymorphic, and fl uses type variables *, **, ***, etc. to represent their types.

File | Interrupt | Help

```
: let f x = hd x;
f::(* list)->*
: letrec map2 fn [] [] = []
 /\    map2 fn (a:as) (b:bs) = (fn a b) : (map2 fn as bs)
 /\    map2 fn _ _ = error "Lists of different length in map2"
;
map2::(*->**->***)->(* list)->(** list)->(*** list)
: letrec fact n = n=0 => 1 | n*fact (n-1);
fact::int->int
: let binom n k = (fact n) / ((fact k) * (fact (n-k)));
binom::int->int->int
: map2 binom [8,8,8,8,8,8,8,8,8] [0,1,2,3,4,5,6,7,8];
[1,8,28,56,70,56,28,8,1]
it::int list
:
```

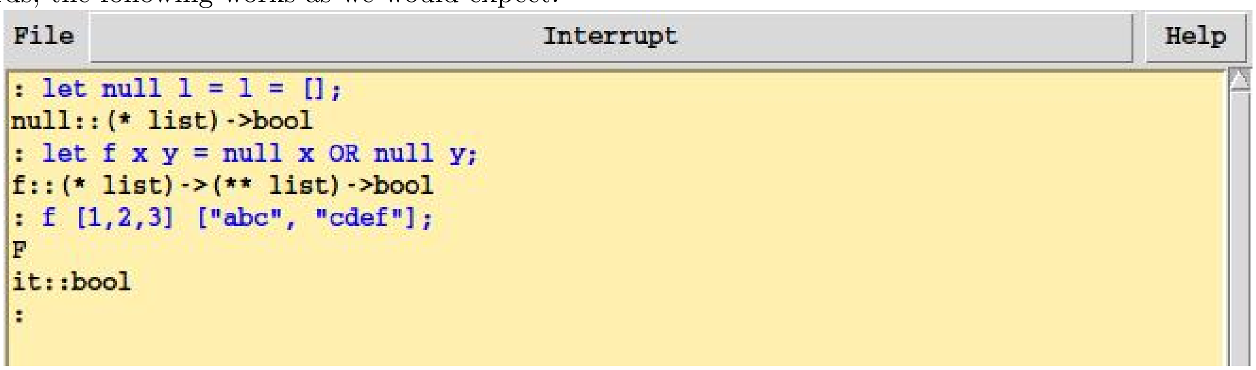
The fl function map2 takes a function f (with argument types * and ** result type ***), and two lists l1 (of elements of type *) and l2 (of elements of type **), and returns the list obtained by applying f to each pair of elements of l1 and l2. Map2 can be used at any instance of its type: above, *, **, and *** were instantiated to int;

below, * is instantiated to (int list) and ** to bool. Notice that the instance need not be specified; it is determined by the type checker.



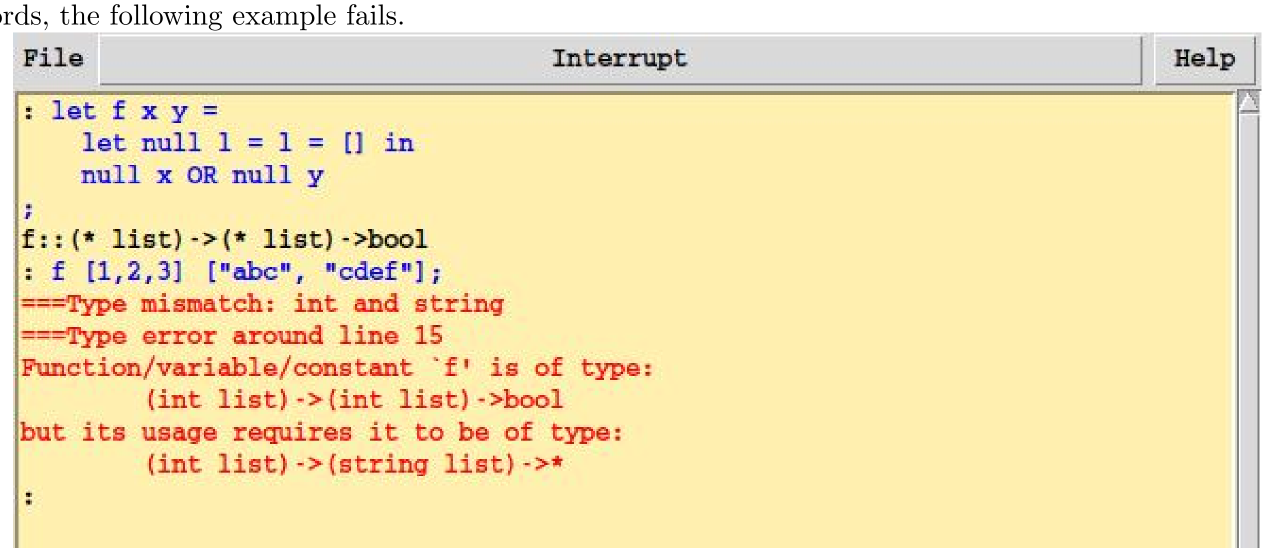

```
: let capitalize s = (chr ((ord (string_hd s))+(ord "A")-(ord "a")))^(string_tl
s);
capitalize::string->string
: let classify name type = sprintf "%s is a %s" (capitalize name) type;
classify::string->string->string
: map2 classify ["john", "anna", "betsy", "bob"] ["boy","girl","girl","boy"];
["John is a boy","Anna is a girl","Betsy is a girl","Bob is a boy"]
it::string list
:
```

It should be pointed out that fl has a polymorphic type system that is sightly different from standard ML's. In particular, only "top-level" user-defined functions can be polymorphic. In other words, the following works as we would expect.




```
: let null l = l = [];
null::(* list)->bool
: let f x y = null x OR null y;
f::(* list)->(** list)->bool
: f [1,2,3] ["abc", "cdef"];
F
it::bool
:
```

However, if we use the same declaration inside the expression, it must be monomorphic. In other words, the following example fails.




```
: let f x y =
    let null l = l = [] in
    null x OR null y
;
f::(* list)->(* list)->bool
: f [1,2,3] ["abc", "cdef"];
===Type mismatch: int and string
===Type error around line 15
Function/variable/constant `f' is of type:
        (int list)->(int list)->bool
but its usage requires it to be of type:
        (int list)->(string list)->*
:
```

In this respect, fl is similar to the functional language called Miranda[1] [40].

1.11 Type Annotations

Sometimes it is useful to inform the type inference mechanism of fl what type is expected. This is particularly useful when there is an obscure type error in the expression you are trying to define.

[1] Miranda is a trademark of Research Software Ltd.

Annotating expressions with the type one expects them to have is often a very efficient method for finding the problem. This is particularly common when using overloaded functions.

In fl a variable or expression can be annotated with its expected type by enclosing it in curly braces and decorate the expression with a type expression. For example, if we would like to define a function ihd that return the head of a list, but that only can be used on integer lists, we could define ihd as follows:

```
: let ihd {l::int list} = hd l;
ihd::(int list)->int
:
```

A more subtle example is the following. Assume we have overloaded the operator + to either operate over strings or integers. We then want to extend this overloading with yet another function over bools. This can be done as:

```
: let iplus {a::int} {b::int} = a+b;
iplus::int->int->int
: let splus {a::string} {b::string} = a^b;
splus::string->string->string
: overload + iplus splus;
: let foo a b = a OR b;
foo::bool->bool->bool
: overload + + foo;
: infix 2 +;
: 1+2;
3
it::int
: "a"+"bc";
"abc"
it::string
: T+F;
T
it::bool
:
```

If we now were to write a function that applies the + function to three arguments, we could accomplish this by defining:

```
File                Interrupt                Help
: let f x y z = x+y+z;
f::[+,+] *->**->***->****
: f 1 2 3;
6
it::int
: f "a" "b" "s";
"abs"
it::string
: f F F T;
T
it::bool
:
```

However, note that the type inferred for f is more general than one might like. After all, all the arguments to f as well as its return type, must have the same type. To capture this, a slightly better definition of f would be:

```
File                Interrupt                Help
: let f {x:: *a} {y:: *a} {z:: *a} = {x+y+z:: *a};
f::[+,+] *->*->*->*
:
```

Here we also demonstrate how polymorphic types can be defined. One warning: make sure there is a space between the :: and the *a so that the parser does not try to look up the symbol ::*a!

1.12 Lambda Expressions

The expression \x.e evaluates to a function with formal parameter x and body e. Thus the declaration let f x = e is equivalent to let f = \x.e. The character \ is our representation of lambda, and expressions like \x.e are called lambda-expressions.

```
File                Interrupt                Help
: \x.x+1;
-
it::int->int
: let q = \x.x+1;
q::int->int
: q 1;
2
it::int
: map (\x.x*x) [1,2,3,4,5];
[1,4,9,16,25]
it::int list
:
```

1.13 Failures

Some standard functions fail at run-time on certain arguments. When this happen, an exception is raised. If that exception is not captured (more below), the failure will propagate to the top-level and an error message will be printed out and any (possibly nested) loads will be aborted. In addition to builtin functions failing, a failure with string "msg" may also be generated explicitly by evaluating the expression error "msg" (or more generally error e where e has type string).

```
File                         Interrupt                          Help
: hd(tl [2]);
Failure:    ---  Cannot compute hd of the empty list
Stack trace:
1:      hd

it::int
: 1/0;
Failure:    ---  Division by zero
Stack trace:
2:      /
1:      i_div

it::int
: error "My message";
Failure:    ---   My message

Stack trace:

it::*
: let select idx l =
    idx < 1 OR idx > length l =>
        eprintf "select called with illegal index (%d)" idx
    |
    el idx l
;
select::int->(* list)->*
: select 10 [1,2,3,4];
Failure:    select called with illegal index (10)
Stack trace:
1:      select

it::int
:
```

A failure can be caught by catch; the value of the expression e_1 catch e_2 is that of e_1, unless e_1 causes a failure, in which case it is the value of e_2. If one wants to examine the error message, one can use gen_catch instead of catch. However, the right-hand side expression to gen_catch must be a function taking the error message as argument. For example:

```
File                              Interrupt                              Help
: let half x = (x % 2) = 1 => eprintf "HALF_ERR: f2 is given an odd number (%d)"
 x
                            | x/2
;
half::int->int
: letrec  robust_half x =
     (half x) gen_catch
     (\msg. str_is_prefix "HALF_ERR" msg => robust_half (x+1) | error msg)
;
robust_half::int->int
: robust_half 2;
1
it::int
: robust_half 3;
2
it::int
: robust_half (1/0);
Failure:    ---    ---  Division by zero
Stack trace:
6:      /
5:      i_div
4:      %
3:      half
2:      gen_catch
1:      robust_half


Stack trace:
2:      gen_catch
1:      robust_half

it::int
:
```

Here we catch only errors with "HALF_ERR" in the error message.

One important property of catch and gen_catch is that they are (very) strict in their first argument. In other words, (hd (e_1 catch e_2)) will completely evaluate e_1 even though only the first element in the list may be needed. In view of fl's lazy semantics, the use of catch should be very carefully considered.

1.14 Boolean Expressions

All Boolean expressions in fl are maintained as ordered binary decision diagrams. Hence, it is very easy to compare complex Boolean expressions and to combine them in different ways. Boolean variables are created by variable s, where s is of type string. The system uses name equivalence, and thus

```
File                    Interrupt                    Help
: let v = variable "v";
v::bool
: v=v;
T
it::bool
: variable "v" = variable"v";
T
it::bool
:
```

The constants true and false are denoted T and F respectively. The standard boolean functions are available, i.e., AND, OR, NOT, XOR, and = are all defined for objects of type Boolean. Furthermore, there is a special identity operator == that return true or false depending on whether the two arguments represent the same Boolean function or not.

Note that unless a oreding has been installed explcitly (more below), the variable ordering in the OBDD representation is defined by the order in which each variable function call *gets evaluated*. Since fl is a fully lazy language, and thus the order in which expressions are evaluated is often difficult to predict, it is strongly recommended that each variable declaration is forced to be evaluated before it is being used. Alternatively, one can request that fl re-orders the variables by evaluating the function var_order and give as argument a list of variable names. fl will then apply the dynamic variable re-ordering mechanism and enforce that the variables mentioned in the list will be the first arguments and that they will occur in exactly this order.

```
File                    Interrupt                    Help
: let a = variable "a";
a::bool
: let b = variable "b";
b::bool
: a AND b;
a&b
it::bool
: a OR b;
a + b
it::bool
: NOT a AND NOT b AND T;
a'&b'
it::bool
: a = b;
a&b + a'&b'
it::bool
: a == b;
F
it::bool
: (a=b) == (a AND b OR NOT a AND NOT b);
T
it::bool
:
```

The default style for printing Boolean expressions is as a sum-of-products. Since this may require printing an extremely large expression, there is a user-setable limit on how many products that will be printed and the maximum size of a product. For more details how to modify these two parameters, see Section 5.

1.15 Quantifiers

There are several ways of using quantification. The "traditional" !x. e (for all x) and ?x. e (there is an x) can be used as long as the type of x and e is bool. In addition, you can also quantify away a variable in an expression by quant_forall v e or quant_thereis v e.

```
File                         Interrupt                          Help
: !a. ?b. (a XOR b);
T
it::bool
: let a = variable "a";
a::bool
: let b = variable "b";
b::bool
: let c = variable "c";
c::bool
: a AND b OR c;
a&b + c
it::bool
: Quant_forall ["a"] (a AND b OR c);
c
it::bool
: Quant_thereis ["a","c"] (a AND b OR c);
T
it::bool
:
```

In fact, quant_forall and quant_thereis quantifies away all variables in the first Boolean expression. For example:

```
: let v s = variable s;
v::string->bool
: let a = v "a";
a::bool
: let b = v "b";
b::bool
: let c = v "c";
c::bool
: let d = v "d";
d::bool
: let ex = (a AND NOT b);
ex::bool
: ex;
a&b'
it::bool
: let nex = ex AND (a=c) AND (b=d);
nex::bool
: Quant_thereis ["a","b"] nex;
c&d'
it::bool
:
```

Note that the actual Boolean expression used as first argument is irrelevant. The only important fact is on what variables the expression depends.

1.16 Dependencies

Sometimes it is useful to find out which Boolean variables a Boolean function actually depends on. The built-in function depends takes a list of elements of type bool and return the union of the variables these functions depend on. For example:

```
File                    Interrupt                    Help
: let v s = variable s;
v::string->bool
: let a = v "a";
a::bool
: let b = v "b";
b::bool
: let c = v "c";
c::bool
: let d = v "d";
d::bool
: let ex1 = (a=c) AND d;
ex1::bool
: let ex2 = a = b;
ex2::bool
: depends [ex1];
["a","c","d"]
it::string list
: depends [ex1,ex2];
["a","b","c","d"]
it::string list
:
```

Note that the order of the variables in the list returned by depends is the variable order of the OBDD representation.

1.17 Substitutions

Given a Boolean function represented as an OBDD, it is convenient to be able to apply the function to some arguments. This can be accomplished by the substitute command that takes a list of (variable name, expression) and an expression in which the simultaneous substitution is to be made. For example,:

```
: let v s = variable s;
v::string->bool
: let a = v "a";
a::bool
: let b = v "b";
b::bool
: let c = v "c";
c::bool
: let d = v "d";
d::bool
: let ex = (a AND NOT b);
ex::bool
: ex;
a&b'
it::bool
: substitute [("a", c), ("b", d)] ex;
c&d'
it::bool
:
```

It should be pointed out that there are no restrictions on the expressions in the substitutions. In particular, it is possible to "swap" variables. We illustrate this by continuing the example above:

```
: ex;
a&b'
it::bool
: substitute [("a", b), ("b", a)] ex;
a'&b
it::bool
:
```

1.18 Type Abbreviations

Types can be given names:

```
: new_type_abbrev pair = int#int;
: let p = (1,2);
p::int#int
:
```

However, as can be seen from the example, the system does not make any distinction between the new type name and the actual type. It is purely a short hand that is useful when defining concrete types below.

1.19 Concrete Types

New types (rather than mere abbreviations) can also be defined. Concrete types are types defined by a set of constructors which can be used to create objects of that type and also (in patterns) to decompose objects of that type. For example, to define a type card one could use the construct type:

File Interrupt Help

```
: lettype card = king | queen | jack | other int;
king::card
queen::card
jack::card
other::int->card
read_card::string->card
write_card::string->card->bool
:
```

Such a declaration declares king, queen, jack and other as constructors and gives them values. The value of a 0-ary constructor such as king is the constant value king. The value of a constructor such as other is a function that given an integer value n produces other(n).

File Interrupt Help

```
: king;
-
it::card
: other 9;
-
it::card
:
```

Note that there is no print routine for concrete types. If a print routine is desired, one has to define it. To define functions that take their argument from a concrete type, we introduce the idea of pattern matching. In particular

```
let f pat1 = e1
 /\ f pat2 = e2
 /\ ...
 /\ f patn = en;
```

denotes a function that given a value v selects the first pattern that matches v, say pati, binds the variables of pati to the corresponding components of the value v and then evaluates the expression ei. We could for example define a print function for the cards in the following way:

```
: let pr_card king = "K"
  /\  pr_card queen = "Q"
  /\  pr_card jack = "J"
  /\  pr_card (other n) = int2str n;
pr_card::card->string
: pr_card king;
"K"
it::string
: pr_card queen;
"Q"
it::string
: pr_card jack;
"J"
it::string
: pr_card (other 5);
"5"
it::string
:
```

If we now issue the top-level command

```
: install_print_function pr_card;

:
```

every time we evaluate an expression of type card this routine would be called and the string printed out on standard out.

Although pattern matching is often sufficient, sometimes one needs to match on some predicate, rather than just the type constructor. For this case, fl provides an `assuming` keyword. For example:

```
: letrec collatz 1 = [1]
 /\    collatz n assuming (n % 2 = 0) = n:(collatz (n/2))
 /\    collatz n assuming (n % 2 = 1) = n:(collatz (3*n+1))
;
collatz::int->(int list)
: collatz 7;
[7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1]
it::int list
:
```

Mutually recursive types can also be defined. To do so, use the keyword andlettype for the subsequent types. For example:

```
: lettype    IExpr = Ivar string | Plus IExpr IExpr | ITE BExpr IExpr IExpr
andlettype BExpr = And BExpr BExpr | GEQ IExpr IExpr;
And::BExpr->BExpr->BExpr
GEQ::IExpr->IExpr->BExpr
read_BExpr::string->BExpr
write_BExpr::string->BExpr->bool
Ivar::string->IExpr
Plus::IExpr->IExpr->IExpr
ITE::BExpr->IExpr->IExpr->IExpr
read_IExpr::string->IExpr
write_IExpr::string->IExpr->bool
:
```

defines two mutually recursive concrete data types for integer expressions and Boolean expressions (very simple versions!).

Currently, fl does not provide any direct way of defining mutually recursive functions. The easiest work-around is to pass the later defined functions as parameters to the earlier function. After all functions have been defined, one can re-define the early ones. To illustrate the approach, consider writing functions that converts objects of type IExpr and BExpr to strings. One possible solution is as follows:

```
: let prIExpr prBexpr expr =
    letrec prIExpr (Ivar s) = s
     /\    prIExpr (Plus a b) = (prIExpr a)^" + "^(prIExpr b)
     /\    prIExpr (ITE c t e) = "if "^(prBexpr c)^" then "^
                                  (prIExpr t)^" else "^(prIExpr e) in
    prIExpr expr
;
prIExpr::(BExpr->string)->IExpr->string
: letrec prBExpr (And a b) = (prBExpr a) ^ " AND " ^ (prBExpr b)
   /\    prBExpr (GEQ a b) = (prIExpr prBExpr a)^" >= "^
                             (prIExpr prBExpr b)
;
prBExpr::BExpr->string
: let prIExpr e = prIExpr prBExpr e;
prIExpr::IExpr->string
:
```

Note that we simply pass prBExpr as an argument to the initial definition of prIExpr.

A slightly easier approach is to use the forward_declare mechanism. With this, we get:

```
File                         Interrupt                          Help
: forward_declare {prBexpr::BExpr->string};
: letrec prIExpr (Ivar s) = s
 /\    prIExpr (Plus a b) = (prIExpr a)^" + "^(prIExpr b)
 /\    prIExpr (ITE c t e) = "if "^(prBexpr c)^" then "^
                              (prIExpr t)^" else "^(prIExpr e)
;
prIExpr::IExpr->string
: letrec prBExpr (And a b) = (prBExpr a) ^ " AND " ^ (prBExpr b)
 /\    prBExpr (GEQ a b) = (prIExpr a)^" >= "^ (prIExpr b)
;
prBExpr::BExpr->string
:
```

which is much easier to write, allthough it requires one to declare the type of the forward function(s).

1.20 Abstract Types

In fl one can also hide the definitions of types, type constructors, and functions. By enclosing a sequence of type declarations and function definitions within begin_abstype end_abstype elist, only the constructors and/or functions mentioned in the elist will be visible and accessible for other functions and definitions. Thus, one can protect a concrete type and only make some abstract constructor functions available. To illustrate the concept, consider defining a concrete type called theorem. The only way we would like the user to be able to create a new theorem is to give a Boolean expression that denotes a tautology (something always true). First we define the basic expression type.

```
File                         Interrupt                          Help
: lettype expr =  E_FORALL string expr |
                 E_VAR string |
                 E_TRUE |
                 E_AND expr expr |
                 E_NOT expr
;
E_FORALL::string->expr->expr
E_VAR::string->expr
E_TRUE::expr
E_AND::expr->expr->expr
E_NOT::expr->expr
read_expr::string->expr
write_expr::string->expr->bool
:
```

For convenenience we define a (simple) pretty printer and install it.

```
: let Pexpr expr =
    letrec do_print indent (E_FORALL s e) =
        (printf "(E_FORALL %s\n%*s" s (indent+2) "") fseq
        (do_print (indent+2) e) fseq
        (printf "%*s)\n" indent "")
     /\    do_print indent (E_VAR s) = printf "%s\n" s
     /\    do_print indent (E_TRUE) = printf "T\n"
     /\    do_print indent (E_AND e1 e2) =
        (printf "(E_AND\n%*s" (indent+2) "") fseq
        (do_print (indent+2) e1) fseq
        (printf "%*s" (indent+2) "") fseq
        (do_print (indent+2) e2) fseq
        (printf "%*s)\n" indent "")
     /\    do_print indent (E_NOT e) =
        (printf "(E_NOT\n%*s" (indent+2) "") fseq
        (do_print (indent+2) e) fseq
        (printf "%*s)\n" indent "")
    in
    (do_print 0 expr) fseq
    ""
;
Pexpr::expr->string
: install_print_function Pexpr;

: let e = E_FORALL "a" (E_AND (E_VAR "a") (E_NOT (E_AND E_TRUE (E_VAR "a"))));
e::expr
: e;
(E_FORALL a
  (E_AND
    a
    (E_NOT
      (E_AND
        T
        a
      )
    )
  )
)

it::expr
:
```

To make it more conveneient to input expressions, one usually defines operators and give them suitable fixities.

File Interrupt Help

```
: let ~ e = E_NOT e;
~::expr->expr
: prefix 0 ~;
: let && a b = E_AND a b;
&&::expr->expr->expr
: infix 4 &&;
: let || a b = E_NOT (E_AND (E_NOT a) (E_NOT b));
||::expr->expr->expr
: infix 3 ||;
: let forall fn s = E_FORALL s (fn (E_VAR s));
forall::(expr->expr)->string->expr
: binder forall;
: let thereis fn s = E_NOT (E_FORALL s (E_NOT (fn (E_VAR s))));
thereis::(expr->expr)->string->expr
: binder thereis;
: let ^^ a b = ~a && ~b || a && b;
^^::expr->expr->expr
: infix 4 ^^;
: // Example
let e = forall a b. thereis c. ~(a^^b) || ~(a^^(c^^b));
e::expr
:
```

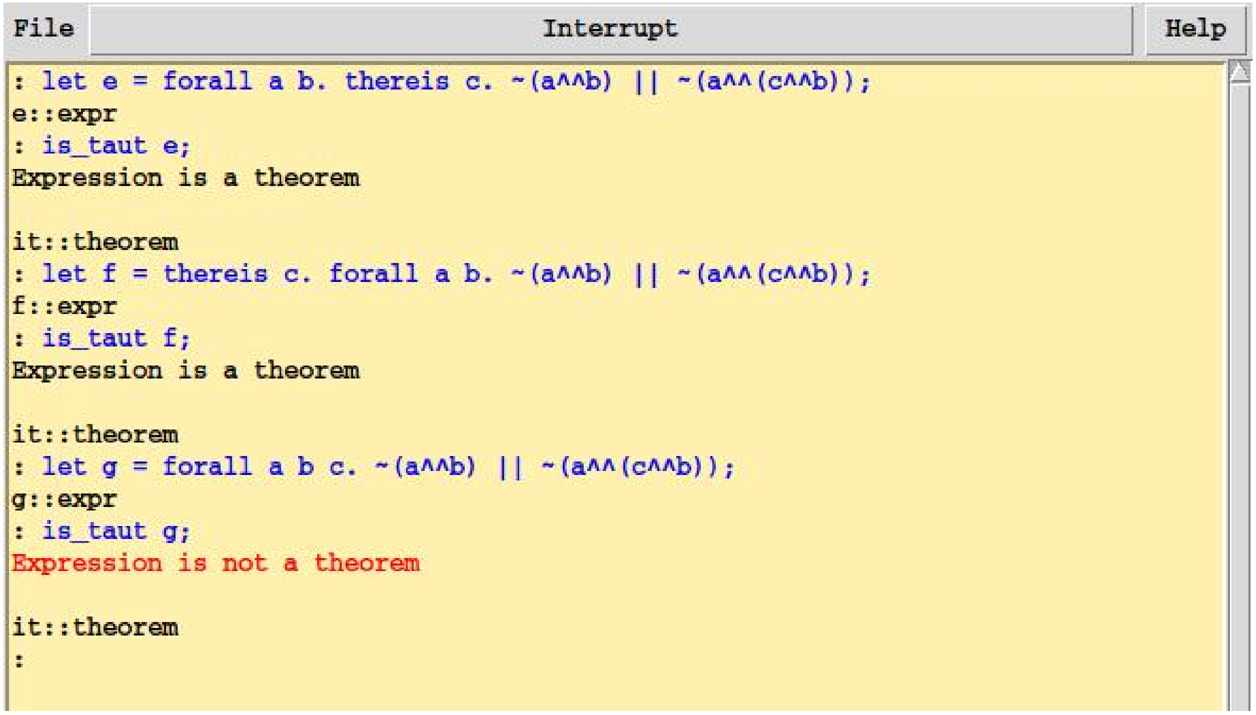
We then define the concrete type theorem and the constructor function is_taut. Note that we also define a couple of help functions. However, only the is_taut function is exported out of the abstract type, and thus is the only way of creating a theorem.

File Interrupt Help

```
: begin_abstype;
: let empty_state = [];
empty_state::* list
: let add_to_state state var value = (var,value):state;
add_to_state::(*#** list)->*->**->(*#** list)
: let lookup var state =
    (assoc var state) catch
    eprintf "Cannot find variable %s (not bound?)" var
;
lookup::string->(string#* list)->*
: let expr2bool e =
    letrec eval (E_FORALL s e) state =
            let state0 = add_to_state state s T in
            let state1 = add_to_state state s F in
            (eval e state0) AND (eval e state1)
     /\    eval (E_VAR s) state = lookup s state
     /\    eval (E_TRUE) state = T
     /\    eval (E_AND e1 e2) state = (eval e1 state) AND (eval e2 state)
     /\    eval (E_NOT e) state = NOT (eval e state)
    in
    eval e empty_state
;
expr2bool::expr->bool
: lettype theorem = THM expr | OTHER;
THM::expr->theorem
OTHER::theorem
read_theorem::string->theorem
write_theorem::string->theorem->bool
: let Ptheorem t =
    let PP (THM e) = fprintf stdout "Expression is a theorem\n"
     /\ PP (OTHER) = fprintf stderr "Expression is not a theorem\n"
    in
    (PP t) fseq ""
;
Ptheorem::theorem->string
: install_print_function Ptheorem;

: let is_taut e = (expr2bool e == T) => THM e | OTHER;
is_taut::expr->theorem
: end_abstype is_taut;
:
```

We can now use this very safe theorem system, since we can only generate theorems that are tautologies. For example



1.21 Infix Operators

In order to make the fl code more readable, it is possible to declare a function to be infix (associating from the left), infixr (associating from the right), nonfix (no fixity at all), prefix (prefix operator with tighter binding than "normal" function definitions), postfix, or of a binder type. For the infix and infixr directives, the precedence can be given as a number from 1 to 9, where a higher number binds tighter. Similarly, prefix also takes a precedence number, but only 0 or 1. Note that prefix and postfix functions bind higher than any infix function. Beware that the fixity declaration modifies the parser and thus remains in effect whether the function is exported out of an abstract data type or note. As an illustration of this idea, consider the following example:

```
File                          Interrupt                          Help
: lettype expr = Val int |
               Mult expr expr |
               Plus expr expr |
               Negate expr;
Val::int->expr
Mult::expr->expr->expr
Plus::expr->expr->expr
Negate::expr->expr
read_expr::string->expr
write_expr::string->expr->bool
: letrec eval (Val i) = i
  /\   eval (Mult e1 e2) = (eval e1) * (eval e2)
  /\   eval (Plus e1 e2) = (eval e1) + (eval e2)
  /\   eval (Negate e1) = 0-(eval e1)
;
eval::expr->int
: let ** a b = Mult a b;
**::expr->expr->expr
: let ++ a b = Plus a b;
++::expr->expr->expr
: infix 4 **;
: infix 3 ++;
: let ' i = Val i;
'::int->expr
: prefix 0 ';
: let q = '1 ++ Negate '2 ** Negate '4;
q::expr
: eval q;
9
it::int
:
```

The next example illustrates how postfix declarations can make the code more readable.

```
File                          Interrupt                          Help
: let ns i = 1000*i;
ns::int->int
: postfix ns;
: let to a b = (a,b);
to::*->**->(*#**)
: infix 3 to;
: 2 ns to 4 ns;
(2000,4000)
it::int#int
:
```

Our final example deals with more advanced binder declarations. The command binder takes a function and makes it into a binder, i.e., an object that introduces a new bound variable in an expression. Note that the type of the function declared to be a binder must be (*→**)→string→**, since the first argument of a binder function will be a lambda expression and the second argument will be a string with the name of the bound variable. Thus, if a function f has been declared as a binder, then f x.E will be parsed as f (\ x.E) "x".

```
File                    Interrupt                    Help
: // Syntactic sugaring
let forall fn s = E_FORALL s (fn (E_VAR s));
forall::(expr->expr)->string->expr
: binder forall;
: let ' v = E_VAR v;
'::string->expr
: free_binder ';
: let && = E_AND;
&&::expr->expr->expr
: infix 4 &&;
: let || a b = E_NOT (E_AND (E_NOT a) (E_NOT b));
||::expr->expr->expr
: infix 3 ||;
: 'a && 'b;
(E_AND
  a
  b
)

it::expr
: forall a b c. a || b && c;
(E_FORALL a
  (E_FORALL b
    (E_FORALL c
      (E_NOT
        (E_AND
          (E_NOT
            a
          )
          (E_NOT
            (E_AND
              b
              c
            )
          )
        )
      )
    )
  )
)

it::expr
:
```

1.22 Overloading

fl supports a limited amount of user defined overloading of functions and operators. However, in order to avoid an exponential type inference algorithm, the overloaded operators must be resolved from the types of their arguments only. To illustrate the construct, consider the following example:

File Interrupt Help

```
: let pplus ({a::int},{b::int}) (c,d) = (a+c, b+d);
pplus::(int#int)->(int#int)->(int#int)
: let pprod ({a::int},{b::int}) (c,d) = (a*c, b*d);
pprod::(int#int)->(int#int)->(int#int)
: overload + + pplus;
: overload * * pprod;
: (1,2)+(3,4)*(9,3*2);
(28,26)
it::int#int
:
```

Here we overloaded the symbols + and *. Note that we essentially added new meanings to + and * since we included the (built-in) versions as possible candidates.

Finally, overloaded operators and functions can of course also be declared infix, binders, or postfix as any other function or operator.

1.23 Quotation of Expressions

Sometimes it is convenient to retain the actual text that was used to denote an expression. By enclosing the expression in ‘‘ and ’’, fl will return a pair consisting of the text of the expression as well as the expression itself.

File Interrupt Help

```
: `` 1+3*4 '';
(" 1+3*4 ",13)
it::string#int
:
```

Note that it is advisable to include an extra space after the opening quotes and before the closing quotes since the parser otherwise is likely to view the quotes as being part of an operator or variable.

Part I

Reference Manual

2 Syntax Summary

This is a (somewhat edited) version of the parser for fl. Since fl is still evolving, the actual syntax accepted by the program may differ slightly from this one. However, I have tried to make the grammar as close as possible to the parser in Voss 1.8b.

```
/* Top level commands */

pgm               : pgm ';' stmt
                  | stmt
                  ;

stmt              : expr
                  | decl
                  | type_decl
                  | 'install_print_function' expr
                  | '' simple_type '' decl
                  | 'print_all_fns'
                  | 'postfix' var_or_infix
                  | 'nonfix' var_or_infix
                  | 'binder' var_or_infix
                  | 'infix' NUMBER var_or_infix
                  | 'infixr' NUMBER var_or_infix
                  | 'begin_abstype'
                  | 'end_abstype' var_or_infix_list
                  | /* Empty */
                  ;

var_or_infix      : VART
                  | INFIX_VAR
                  | INFIXR_VAR
                  | POSTFIX_VAR
                  | BINDER_VAR
                  ;

var_or_infix_list : var_list var_or_infix
                    | var_or_infix
                    ;
```

```
/* Declarations */
decl              : 'let' fn_defs
                  | 'letrec' fn_defs
                  ;

fn_defs           : fn_def '/'_ fn_defs
                  | fn_def
                  ;
fn_def            : var_or_infix lhs_expr_list
                  | '(' var_or_infix lhs_expr_list ')'
                  ;
lhs_expr_list     : expr1 lhs_expr_list
                  | '=' top_expr
                  ;
```

```
/* Type declarations */
type_decl       : 'lettype' type_name '=' type_expr_list
                | 'new_type_abbrev' var_or_infix '=' simple_type
                ;
type_name       : type_name ',' var_or_infix
                | var_or_infix
                ;
type_expr_list  : type_expr_list ',' type_expr
                | type_expr
                ;
type_expr       : type_expr '|' type
                | type
                ;
type            : var_or_infix type_list
                ;

type_list       : type_list simple_type
                | /* Empty */
                ;
simple_type     : var_or_infix
                | simple_type '->' simple_type
                | simple_type '#' simple_type
                | simple_type 'list'
                | '(' simple_type ')'
                ;
```

```
/* Expressions */
top_expr        : expr ;

expr            : decl ’in’ expr
                | ’val’ expr1 ’=’ expr ’in’ expr
                | ’_’ expr1 ’.’ expr
                | BINDER_VAR expr1 ’.’ expr
                | expr ’=>’ expr ’|’ expr
                | expr POSTFIX_VAR
                | expr INFIX_VAR expr
                | expr INFIXR_VAR expr
                | expr ’,’ expr
                | expr ’=’ expr
                | expr05
                ;
expr05          : expr05 expr1
                | expr1
                ;

expr1           : ’[’ expr_list ’]’
                | ’[’ ’]’
                | ’(’ expr ’)’
                | expr2
                ;

expr_list       : rev_expr_list
                ;

rev_expr_list   : rev_expr_list ’,’ expr
                | expr
                ;

expr2           : VART
                | NUMBER
                | ’[]’
                | ’" ... "’
                | ’quit’
                ;
```

2.1 Reserved Words in fl

The following list contains all identifiers that are currently defined in fl. This list will likely change in future releases.

> **begin_abstype binder end_abstype forall_last HOL_DEF HOL_EXPR infix infixr in install_print_function let letrec lettype HOLlettype list new_type_abbrev nonfix overload postfix val quit**

In addition, the following symbols are also defined and cannot be redefined:

> **=> -> | = # /\ \ ::**

3 Buit-in Functions and Commands

The following subsections contain all built-in functions and top-level comamands in fl. Most of them are described in the tutorial section, but this section should be viewed as the reference section.

3.1 Functions

Note that all these functions can be redefined by the user or by library files, and thus the table is only valid for “raw” fl. The functions are given in alphabetical order.

bool2str

Type: *bool→string*
Usage: **bool2str f**
Convert the Boolean function `f` to a Boolean expression and return a string that prints out this expression. Note that the expression can be truncated if the size of the Boolean expression exceeds some threshold. For more details, see the section on printing Boolean functions on page **??**.

bdd_size

Type: *(bool list)→int*
Usage: **bdd_size [f1,f2,...,fn]**
Return the number of OBDD nodes needed to represent the functions f1, f2, ..., fn. Note that bdd_size[f1]+bdd_size[f2] is usually significant larger than bdd_size[f1,f2], due to sharing in the OBDD structure.

bdd_load

Type: *string→(bool list)*
Usage: **bdd_load FileName**
Read in the OBDD functions from file FileName. This fucntion assumes the OBDDs were saved in the format used by bdd_save.

bdd_reorder

Type: *int→bool*
Usage: **bdd_reorder n**
Invoke the dynamic variable re-ordering routine in order to reduce the size of the OBDDs. The parameter n denote the number of times the re-ordering should be done.

bdd_save

Type: *string→((bool list)→bool)*
Usage: **bdd_save FileName [f1,f2,...,fn]**
Save the OBDDs rooted at f1, f2, ..., fn. The format of the saved file is in itself a valid fl program and thus is fairly self-explanatory.

chr

Type: *int→string*
Usage: **chr n**
Return the string (of length 1) of the character corresponding to the ASCII code n. If n is less than 0 or greater than 127, the function fails with a ranger error message.

depends

Type: *(bool list)→(string list)*

Usage: **depends [f1, f2, ..., fn]**
Return the list of names of Boolean variables that the functions f1, f2, ..., fn depends on. Note that the order of the variables in the string correspond to the OBDD variable ordering.

error
Type: *string→failure*
Usage: **error msg**
Raise an exception with error message msg.

empty
Type: *(*list)→bool*
Usage: **empty l**
Return T (true) if list l is empty. Otherwise return F (false).

explode
Type: *string→(string list)*
Usage: **explode s**
Convert the string s into a list of strings of length 1.

eval
Type: *string→bool)*
Usage: **eval s**
Eval writes out the string s on a temporary file, redirects fl's standard input to this file and returns T if this was succsessful. Otherwise it return F and leaves the standard input as before. Note that this function is most useful at the top level where it can be used to implement reference variables.

fanin
Type: *fsm→(string→(string list))*
Usage: **fanin ckt n**
For fsm object ckt, return the list of node names that the next state function for node s depends on.

fanout
Type: *fsm→(string→(string list))*
Usage: **fanout ckt n**
For fsm object ckt, return the list of node names whose next state functions depend on the value of node s. Note that fanout uses the topology of ckt to detremine fanout. Thus, it is possible that the function is conservative and returns node names that, functionally speaking, are not in the fanout set of node s.

STE
Type: *string→fsm→(fourtuple list) →(fivetuple list)→(fivetuple list)→((string#int#int) list)→bool where fourtuple is bool#string#int#int and fivetuple is bool#string#bool#int#int*
Usage: **STE options ckt ant cons tracenodes**
Perform symbolic trajectory evaluation on the circuit ckt. The options is a string that give various options to the symbolic trajectory evaluator. Currently recognized options are:

-a Abort the verification at the first antecedent or consequent failure. If the verification is aborted, STE will return a Boolean function that gives *the condition for this failure to manifest itself.* Note that this is contrary to STE's usual behavior which is to return the Boolean function that gives the conditions for the verification to succeed.

-m n Abort the verification after reaching time n.

-i Allow antecedent failures. In other words, compute a straight implication. The normal behavior of the verification process is to disallow antecedent failures. Thus the default verification condition is both to check that every trajectory the circuit can go thorough that is consistent with the antecedent is also consistent with the consequent, and that there is at least one (real) circuit trajectory that is consistent with the antecedent.

-w Do not print out warning messages.

-t s In addition to printing out trace messages on stderr, also send the trace events in Postscript format to the file s. By previewing or printing out the file the user gets a waveform diagram for the traced signals.

-T s Same as -t, but generate Postscript code in landscape mode.

For more details about symbolic trajectory evaluation, see Section **??**.

fst

Type: *(*#**)→**
Usage: **fst obj**
Return the first part of a pair.

get_node_val

Type: *fsm→(string→(bool#bool))*
Usage: **get_node_val fsm nd**
Return the current value on node nd in the model structure fsm. This function is mostly useful after evaluating an STE command using the -a (abort) or -m (maximum number of steps) options. The pairs of Boolean functions returned represent the high and low values respectively. As usual, the encoding is 0=(F,T), 1=(T,F), X=(T,T), and top=(F,F).

get_delays

Type: *fsm→(string→((int#int)#(int#int)))*
Usage: **get_delays fsm nd**
Return the tuple ((min-rise-delay, max-rise-delay), (min-fall-delay, max-fall-delay)) for the node nd in the model structure fsm.

hd

Type: *(* list)→**
Usage: **hd l**
Return the head (first element) of a list.

implode

Type: *(string list)→string*
Usage: **implode sl**
Concatenate the strings in sl and return the resulting string. Note that there is a limit on how big a string can be. Currently this limit is around 16,000 character. Note also, that currently the strings are not garbage collected. However, all strings are made unique internally and thus only the total number of distinct strings determines the memory usage for the strings.

int2str

Type: *int→string*
Usage: **int2str i**
Returns a string version of the integer i. If i is negative the string will begin with a minus sign.

is_stable

Type: *fsm→bool*

Usage: **is_stable ckt**

Returns true if the circuit ckt is currently stable, i.e., no the value on each node equals its excitation. This function is mostly useful after evaluating an STE command using the -a (abort) or -m (maximum number of steps) options.

load

Type: *string→bool*

Usage: **load fn**

Re-direct standard input to read from the file fn until the file is exhausted at which point it is restored to its current source. Returns T (true) if the file can be found. Returns F otherwise. If an error occurs during reading a file, all nested reads are terminated and the system returns to the user level prompt.

load_exe

Type: *string→fsm*

Usage: **load_exe fn**

Loads an fsm object described in the .exe format stored in the file named fn. Note that node delays can be provided in a companion file (with same name but with suffix .del).

make_fsm

Type: *Set→fsm*

Usage: **make_fsm cl**

Converts a next state function defined as the concrete data type Set to the internal fsm object. For more details on the Set data type, see Section **??**.

NOT

Type: *bool→bool*

Usage: **NOT f**

Returns the negation of the Boolean function f.

nodes

Type: *fsm→((string list) list)*

Usage: **nodes fsm**

Returns the list of nodes that the model structure fsm contains. For each node, the list of names (including aliases) are returned.

ord

Type: *string→int*

Usage: **ord s**

Returns the ASCII code for the first character in the string. The function fails with a length constraint message if it is not given a string of length 1.

print

Type: *string→string*

Usage: **print s**

Prints out the string s on standard output and returns an emtpy string. Note that due to fl's lazy semantics, constants will usually only be printed out once. For example,

```
let ans = print "OK"; ans; ans; ans;
```

will only print out "OK" once, rather than three times. The usual work-around is to catenate constant strings with some variable string and then print out the final string.

print_fsm
Type: *fsm→string*
Usage: **print_fsm fsm**
Prints out the internal representation of a model structure. For more details of the format, see Sections **??**.

profile
Type: *(bool list)→(int list)*
Usage: **profile l**
Returns the widths of the OBDD multigraph indicated by the list of Boolean functions in l. Note that this routines correctly accounts for sharing. If individual function sizes are needed, call profile with singelton lists. User's rarely call this function directly, but rather calls the function bdd_profile defined in the defaults.fl file which prints out a graphical profile of the OBDD multigraph.

rvariable
Type: *string→bool*
Usage: **rvariable s**
Returns a random Boolean value. Note that there is only one value associated with each variable name s. Thus, (rvariable "a") == (rvariable "b") always holds.

quant_forall
Type: *bool→(bool→bool)*
Usage: **quant_forall vf f**
Returns the result of universally quantifying out all variables occuring in the Boolean function vf from the Boolean function f.

quant_thereis
Type: *bool→(bool→bool)*
Usage: **quant_thereis vf f**
Returns the result of existentially quantifying out all variables occuring in the Boolean function vf from the Boolean function f.

snd
Type: *(*#**)→***
Usage: **snd obj**
Returns the second part of a pair.

save_exe
Type: *fsm→(string→bool)*
Usage: **save_exe fsm s**
Save the model structure fsm in the exe format in file s. Returns true if it is successful and false otherwise.

substitute
Type: *((string#bool) list)→(bool→bool)*
Usage: **substitute sl f**
Simultaneously perform the substitutions given in the substitution list sl in the Boolean

function f. A substitution in the list is a pair–variable to be substituted and boolean function to be substituted in. Formally, given the substitution list sl = $[(v_1, f_1), \ldots, (v_n, f_n)]$, we have substitute sl f = $\exists v_1. \ldots \exists v_n. \; ((v_1 = f_1) \wedge \ldots \wedge (v_n = f_n) \wedge f)$.

system

Type: *string→int*
Usage: **system s**
System causes the string s to be given to sh(1) as input as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

tl

Type: *(* list)→(* list)*
Usage: **tl l**
Returns the tail (all but the first element) of a list. Fails and raises an exception on the empty list.

time

Type: **→(*#string)*
Usage: **time e**
Evaluate e and return the pair (e,t), where t is a string containing the usertime used to evaluate e. Note that the laziness of fl may make consecutive invocations require vastly different amount of time. On HP machines, time returns the wall time. This is a "feature" that is likely to be removed soon.

trace

Type: *string→fsm→(fourtuple list) →(fivetuple list)→(fivetuple list)→((string#int#int) list)→bool where fourtuple is bool#string#int#int and fivetuple is bool#string#bool#int#int*
Usage: **trace options ckt ant cons tracenodes**
Performs the same operations as STE (with the same arguments), but returns a list of node changes. For each node in the trace list and for each time t that node change value, a tripple (t,H,L) is added to a list that is returned by the trace command. The H and L are the boolean functions for the high and low value on a node using the standard encoding, i.e., 0 is (F,T), 1 is (T,F), X is (T,T) and overconstrained is (F,F). Note that the command is eager and thus the whole trace is computed before the list is returned. As a result, the command can be very expensive to evaluate since there may be a large number of distinct Boolean functions generated that must be rememberd (and thus not garbage collected).

variable

Type: *string→bool*
Usage: **variable s**
Returns the Boolean function that is T if the Boolean variable s is T and F otherwise.

var_order

Type: *(string list)→bool*
Usage: **var_order l**
Re-orders the Boolean variables so that the variables listed in l will be the first variables in the OBDD ordering.

AND

Type: *bool→(bool→bool)*

Usage: **f AND g**
Computes the AND of two Boolean functions.

OR

Type: *bool→(bool→bool)*
Usage: **f OR g**
Computes the OR of two Boolean functions.

XOR

Type: *bool→(bool→bool)*
Usage: **f XOR g**
Computes the XOR of two Boolean functions.

%

Type: *int→(int→int)*
Usage: **f%g**
Computes f MOD g. Note that fl uses arbitrary precision integers.

*

Type: *int→(int→int)*
Usage: **f*g**
Computes f times g. Note that fl uses arbitrary precision integers.

/

Type: *int→(int→int)*
Usage: **f/g**
Computes f divided by g. Note that fl uses arbitrary precision integers.

+

Type: *int→(int→int)*
Usage: **f+g**
Computes f plus g. Note that fl uses arbitrary precision integers.

-

Type: *int→(int→int)*
Usage: **f-g**
Computes f minus g. Note that fl uses arbitrary precision integers.

:

Type: **→(* list)→(* list)*
Usage: **e:l**
Inserts element e as the first element in the list l.

^

Type: *string→(string→string)*
Usage: **f ^ s**
Catenates strings f and s and return the result. Note that the maximum string length in fl is currently about 16,000 characters and strings longer than that are simply truncated (after a warning message is printed out to standard error).

<

Type: *int→(int→bool)*

Usage: **f<g**
Returns T if f<g and F otherwise.

<=

Type: _int→(int→bool)_
Usage: **f<=g**
Returns T if f<=g and F otherwise.

>

Type: _int→(int→bool)_
Usage: **f>g**
Returns T if f>g and F otherwise.

>=

Type: _int→(int→bool)_
Usage: **f>=g**
Returns T if f>=g and F otherwise.

=

Type: _*→(*→bool)_
Usage: **f=g**
If f and g are Boolean functions, f=g returns the exclusive NOR of the two functions. If f and g are any other functions, the result is T if the two functions have the same name and F otherwise. For other values, f=g returns T if f and g are structurally equal and F otherwise.

==

Type: _*→(*→bool)_
Usage: **f==g**
If f and g are Boolean functions, f==g returns T if the two functions are equal and F otherwise. For other values, f==g and f=g behaves identically.

!=

Type: _*→(*→bool)_
Usage: **f!=g**
If f and g are Boolean functions, f!=g returns the exclusive OR of the two functions. If f and g are any other functions, the result is F if the two functions have the same name and T otherwise. For other values, f!=g returns F if f and g are structurally equal and T otherwise.

catch

Type: _*→*→*_
Usage: **e catch f**
Forces a complete evaluation of e. If e results in an exception f is returned. Otherwise the result of evaluating e is returned.

seq

Type: _*→**→**_
Usage: **e seq f**
Evaluates first e and then f and returns the result of evaluating f.

3.2 Top-level commands

The following set of functions/commands are only available at the top level, i.e., directly after a prompt (if the session had been done interactively). Most of these commands modify the way fl parses input or presents output and are meant to make it easier to embed other languages in fl. As with all such user-defined changes to the parser, havoc can easily break out. Thus, we strongly sugest that careful consideration be given to which operators/functions that will be overloaded, made infix or postfix, and which types gets userdefined print routines.

install_print_function
: Type: *(*→string)→()*
  Usage: **install_print_function f**
  Install_print_function takes a function f of type *→string, where * is a concrete type (defined using lettype) and installs this function as the pretty printer for objects of type *. Thus whenever an object of type * gets evaluated at the top level, fl will automatically call the function f and print out the resulting string. Note that only concrete types can have pretty printer functions associated with them.

binder
: Type: *((*→**)→string→**)→()*
  Usage: **binder f**
  Binder takes a function and makes it into a binder, i.e., an object that can introduce a new bound variable in an expression. Note that the type of the function must be (*→**)→string→**, since the first argument of a binder function will be a lambda expression and the second argument will be a string with the name of the bound variable. Thus, if a function f has been declared as a binder, then f x.E will be parsed as f (\ x.E) "x".

postfix
: Type: *(*→**)→()*
  Usage: **postfix f**
  Postfix takes a name of a function and modifies the parser so that the function will be parsed as a postfix operator. Postfix operators have higher precedence than infix, infixr, and prefix functions.

nonfix
: Type: *(*→**)→()*
  Usage: **nonfix f**
  Nonfix takes a name of a function and removes the special parsing of the function. If the function is not declared as a binder, postfix, infix or infixr operator, nonfix is (essentially) a no-op.

infix
: Type: *(int→*→**)→()*
  Usage: **infix p f**
  Infix takes a number p between 0 and 9 and a function or operator f. The command causes the parser to view f as an infix operator with precedence p that associates to the left.

infixr
: Type: *(int→*→**)→()*
  Usage: **infixr p f**
  Infixr takes a number p between 0 and 9 and a function or operator f. The command causes the parser to view f as an infixr operator with precedence p that associates to the right.

overload

Usage: **overload s f1 f2 f3 ... fn**

Overload takes a name s and a list of names (f1 .. fn) (possibly decorated with type) of functions or operators. From here on, the fl parser will overload the name s to denote anyone of the functions. Note that the oveloading in fl is somewhat restricted, since only the argument types are used in resolving the overloading. As a result, it is often necessary to type the arguments to a function explicitly in order for fl to succeed in resolving the function requested. As usual, with overloading, it is easy to create programs that appears to be correct, but that denotes something completely different. Thus, we strongly suggest the overloading of operators should be used sparingly.

begin_abs_type

Usage: **begin_abs_type;**

Begin_adt starts the definition of a (possibly nested) abstract data type.

end_abs_type

Usage: **end_abs_type f1 f2 ... fn;**

End_abs_type takes a list of functions and/or type constructors to be exported from the abstract data type. Thus of all the functions and operators defined inside the abstract data type, only the ones listed after end_abs_type will be visible to other functions.

set_prompt

Usage: **set_prompt s;**

Change the interactive prompt to the string s.

4 Summary of Predefined Functions

The following list contains all predefined functions in fl. The vast majority of these functions can be re-defined. In the list is also indicated whether the function is infix, whether it associates to the left or right and the precedence of the operator.

String manipulations

chr		Convert an integer to the ASCII character corresponding to it.
ord		Given a string returns the ASCII code for it.
explode		Convert string to list of single character strings.
implode		Takes a list of single character strings and catenates them together.
bool2str		Convert a Boolean to a string.
int2str		Convert integer to string for printing purposes.

General functions

catch	infix 2	Evaluate lhs, if it fails return e2 otherwise return result of lhs.
error		Fail and print out message.
empty		Applied to a list returns true if list is empty, false otherwise.
load		Re-direct standard input to this file;
print		Given a string, prints it out on stdout. Watch out for laziness!
time		Given an expression forces it to be completely evaluated and returns a pair of result, time pair.
seq	infix 1	Evaluate lhs first, throw away result and then evaluate rhs.

Boolean

<	infix 3	Less than.
<=	infix 3	Less than or equal.
==	infix 3	Identical.
!=	infix 3	Not equal.
>	infix 3	Greater than.
>=	infix 3	Greater than or equal to.
variable		Given a string returns the Boolean variable with this name.
AND	infix 4	Boolean conjunction.
OR	infix 3	Boolean disjunction
XOR	infix 3	Boolean exclusive or
NOT		Boolean negation.
!v.e		compute for all x in 0,1 e
?v.e		compute there is x in 0,1 e
bdd_size		Given a list of Boolean functions, returns the total size in number of BDD nodes
depends		Given a list of Boolean functions, returns a list of the Boolean variables the function depends on.
quant_forall		Universally quantify away all Boolean variables in the first argument from the expression in the second argument.
quant_thereis		Existentially quantify away all Boolean variables in the first argument from the expression in the second argument.
substitute		Applies a substitution to a Boolean expression.

Finite State Machine Manipulation

load_exe	Read in exe file and return the fsm.
make_fsm	Converts fl description of system into fsm.
nodes	Given fsm returns a list of node lists. Each node list consists of all aliases for the node.
fanin	Given fsm model and node name returns a list of node names the next state function of the node depends on.
fanout	Given fsm model and node name returns a list of node names the nodes that depend on the value of this node.
get_node_val	Given fsm model and node name returns the encoded version of the current value on the node.
is_stable	Given fsm model returns true (T) if the model is currently stable. Returns false (F) otherwise.
print_fsm	Print out an internal representation of STE. Pretty obscure.
STE	Basic trajectory evaluation function.

Dealing with Cartesian Products

e1 , e2	Returns the tuple (e1, e2)
fst	Returns the first element in tuple.
snd	Returns the second component of tuple.

Dealing with Lists

hd		Returns the first element in a list.
tl		Returns the tail of a list. Note that tl [] = [].
:	infixr 2	Corresponds to the CONS operator in LISP.

Arithmetic Functions

	infix 4	Multiplication.
/	infix 4	Integer division.
+	infix 3	Integer addition.
-	infix 3	Integer subtraction.
^	infix 3	String catenation.

5 The .vossrc Default File

If the user puts a .vosrc file in his/her home directory or in the current directory, fl will read this file to set a number of defaults. Below we include a copy of the default .vossrc file which also include the acceptable alternatives.

```
###########################
# Run time options for fl #
###########################
# Where to search for fl library files
VOSS-LIBRARY-DIRECTORY = /isd/local/generic/lib/vosslib
#
# PRINT-ALIASES: should both primary node name and aliases be printed?
PRINT_ALIASES = TRUE
#
# PRINT-FORMAT for Boolean expressions: SOP (sum-of-products) INFIX TREE
PRINT-FORMAT = SOP
MAX-PRODUCTS-IN-SOP-TO-PRINT = 5
#
PRINT-TIME = TRUE
NOTIFY-OK_A-FAILURES = TRUE
NOTIFY-OK_C-FAILURES = TRUE
NOTIFY-TRAJECTORY-FAILURES = TRUE
NOTIFY-CHECK-FAILURES = TRUE
PRINT-FAILURE-FORMULA = TRUE
#
# Max number of steps to reach stability before setting to X
STEP-LIMIT = 100
#
#
# Max number of errors actually reported during STE
MAXIMUM-NUMBER-ERRORS-REPORTED = 5
#
# DELAY-MODEL is one of: UNIT-DELAY, MINIMUM-DELAY, MAXIMUM-DELAY,
#                        AVERAGE-DELAY, or BOUNDED-DELAY
DELAY-MODEL = UNIT-DELAY
#
# For historical reasons... will likely be removed in next release
DEBUG-MODE = NO
#
# Allow dynamic re-rordering of OBDD variables
DYNAMIC-ORDERING = YES
# Should re-ordering try to find optimal ordering or stop sooner
OPTIMAL-DYNAMIC-ORDERING = YES
# How much is the OBDD table allowed to *grow* during re-ordering
ELASTICITY-IN-DYNAMIC-ORDERING = 2
# How many times should the sifting algorithm be applied?
DYNAMIC-ORDERING-REPETITIONS = 1
#
#
# Should messages be printed for garbage collections and re-orderings
VERBOSE-GARBAGE-COLLECTION = YES
```

6 Standard Libraries

Since Fl is a very young language, there are no extensive standard libraries and thus this section is very tentative and is likely to be modified significantly in future releases. All standard libraries reside in the vosslib directory. The easiest way to make sure this directory is in the search path for fl is to create a file called *.vossrc* in your home directory that contains a line

```
VOSS-LIBRARY-DIRECTORY = /path/where/voss/is/installed/vosslib
```

6.1 defaults.fl

This is a basic library that contains many useful general functions.

length l	Returns the length of the list l.
append l1 l2	Appends lists l1 and l2. Note '@' and 'and' are infix aliases to append.
el i l	Select element i in list l. List elements are numbered from 1.
last l	Return the last element in list l.
butlast l	Return the list l with the last element removed.
replicate x n	Return a list with n copies of x as elements.
map fn l	Apply the function fn to each element of the list l and return the resulting list.
itlist f l x	Combine all the elements in l with the function f, i.e., f (hd l) (f (el 2 l) (f (el 3 l) (... (f (last l) x)))...).
rev_itlist f l x	As itlist, but do it in reverse.
find p l	Return the first element in the list that makes the predicate p true.
exists p l	Determine whether an element exists in the list l that satisfies the predicate p.
forall p l	Determine whether all elements in in the list l satisfies the predicate p.
mem x l	Determines whether there is an element in l equal to x.
assoc x al	Return the second component of a pair in the list l whose first component is equal to x.
rev_assoc x l	Same as assoc but exchange meaning of fst and snd.
rev xl	Reverse list xl.
filter p l	Returns the list obtained by removing from l every element that does not satisfy p.
flat ll	Takes a list of lists and return the list obtained by merging all the lists.
interleave ll	Takes a list of lists and returns a single list that is the interleaving of each list.
combine l r	Takes lists l and r and creates a list of pairs whose first components are drawn from l and whose second components are drawn from r.
split pl	Takes a list of pairs and returns a pair of lists. The first list are all first components of the pairs and the second list contain all second components of the pairs.
s1 intersect s2	Return the list of elements common to both s1 and s2.
s1 subtract s2	Return the list of elements that are in s1 but not in s2.
s1 union s2	Return the union (no duplicates) of s1 and s2.
distinct l	Determines whether there are any duplicates in the list l. If so, returns false; otherwise returns true.
setify l	Take a list and make it into a set (no duplicates).

s1 set_equal s2	Determines whether the two sets s1 and s2 are equal.
declare vl	Takes a list of Boolean variables and forces them to be evaluated in the order they appear in the list. Useful in declaring Boolean variables for the OBDD ordering.
num2str n	Converts a number (positive) to a string.
lg n	Computes the number of bits requires to represent n as an unsigned binary number.
bdd_profile expr_list	Takes a list of Boolean expressions and prints out a histogram over the width of the combined OBDD forest.
excitation ckt nd	Computes the next state function for node nd in circuit ckt. *NOTE:* this function only works correctly when using the UNIT-DELAY model.
node_profile ckt node_list	Prints out a bdd profile for all the current values on the nodes in node_list. Mostly useful in conjunction with the '-m n' option to STE (i.e., abort the simulation at a suitable time and check the size and profile of the OBDDs on the selected nodes.

6.2 verification.fl

This is the basic verification library that contains useful functions to make writing verification conditions much more convenient. It should be noted that this is an abstract data type so not all functions defined in the library are exported. In order to shorten the typing information, we use the following shorthand: voss_tuple = (bool, (string, (bool, (int, int)))). All these functions, with the exception of node_vector, variable_vector, verify and nverify returns lists of five-tuples, of the form described in the description of STE. Briefly, the functions are as follows:

UNC	Unconstrained. Useful as padding when writing functions generating verification conditions.
n is v	Node n is asserted/checked to have the value v with guard true.
nv isv vv	Node list nv is asserted/checked to have the values in the value list vv with guard true.
node_vector s n	Create a list of strings of the form s catenated with the string representing integer i, where i goes from (n-1) to 0.
variable_vector s n	Create a list of Boolean variables of the form s catenated with the string representing integer i, where i goes from (n-1) to 0. Note that these variables are not declared until they are forced to be evaluated. See declare in defaults.fl for a function to do so.
vl when e	Imposes the domain constraint denoted by the Boolean expression e on all the five-tuples in vl.
vl from t	Set all the starting times in the five-tuples in vl to t.
vl to t	Set all the ending times in the five-tuples in vl to t.
during f t vl	Set all the starting and ending times in the five-tuples in vl to f and t respectively.
vl1 then vl2	Merge the lists together, but adjust the durations for the five-tuples in vl2 so that the "time 0" for vl2 is equal to the maximum time of any five-tuple in vl1
vl for t	Same as 'to' above.
verify fsm l ant cons trl	An old shorthand for (declare l) seq (STE "" fsm [] ant cons trl. Probably should be removed.
nverify fsm l ant cons trl	Same as verify but the first argument is the option string to STE. Again, should be viewed as obsolete.

SymbIndex nl addr fn	Symbolic indexing function. Will apply the function fn to every element i in nl and then apply a when condition to each result that requires addr to be equal to i.

6.3 arithm.fl

This is a library of bitvector functions. A bitvector is represented as a list of Booleans and is viewed as a big-endian vector, i.e, the head of the list is the most significant bit.

num2bv sz n	Convert the integer n to a bitvector of size sz.
bv2num bv	If the bitvector bv does not contain any Boolean variables, view it as an unsigned binary number and convert it to a number.
prefix n av	Return the n first elements of av.
suffix n av	Return the n last elements of av
av add bv	Add the two bitvectors together.
increment av	Add one to the bitvector av.
ones_complement av	Return the 1's complement of the bitvector av.
twos_complement av	Return the 2's complement of the bitvector av.
av subtract bv	Subtract bitvector bv from bitvector av.
av greater bv	Compute the Boolean expression for the number denoted by av is greater than the number denoted by bv. Both bitvectors are viewed as unsigned integers.
av equal bv	As for greater, but for equality.
av geq bv	As for greater, but for greater than or equal to.
av less bv	As for greater, but for less than.
av leq bv	As for greater, but for less than or equal to.
av bvAND bv	Bit-wise AND.
av bvOR bv	Bit-wise OR.
av bvXOR bv	Bit-wise XOR.
bvNOT av	Bit-wise NOT.

6.4 HighLowEx.fl

This library defines only two funcitons: Hexpl and Lexpl. The basic task of both is to take a Boolean function and return an assignments to some set of variables that would make the Boolean function evaluate to true. The two functions differ only in that Hexpl tries to find an assignment with as many 1's as possible, whereas Lexpl tries to assign as many 0's as possible. In order to make the output more readable, both functions take as first argument a list of pairs. The first element in the pair is a string and the second argument is a list of Boolean variables. The string will be used as a header for the assignemnts to the list of variables. For example, is i, Aa, Ab, a, b, d, and q denote lists of Booelan variables, and f denote some Boolean expression over these (and possibly other) variables, then we may get:

```
: Lexpl [("I",i),("Aa",Aa),("Ab",Ab),("a",a),("b",b),("d",d),("q",q)] f;
"
I = 001011000
Aa = 0000
Ab = 0001
a = 1111
b = 0000
d = 0000
q = ----

"
```

where 0's and 1's denote assignemnt to the corresponding variables to make f evaluate to 1, whereas $-$ denote don't care conditions.

References

[1] D. L. Beatty, R. E. Bryant, and C.-J. H. Seger, "Synchronous Circuit Verification by Symbolic Simulation: An Illustration," *Sixth MIT Conference on Advanced Research in VLSI*, 1990, pp. 98–112.

[2] D. L. Beatty, "A Methodology for Formal Hardware Verification, with Application to Microprocessors," Technical Report CMU-CS-;93-190, Carnegie Mellon University, 1993.

[3] D. L. Beatty, and R. E. Bryant, "Formally Verifying a Microprocessor using a Simulation Methodology," *31st Design Automation Conference*, June, 1994, pp. 596–602.

[4] S. Bose, and A. L. Fisher, "Verifying Pipelined Hardware Using Symbolic Logic Simulation," *International Conference on Computer Design*, IEEE, 1989, pp. 217–221.

[5] S. Bose, and A. L. Fisher, "Automatic Verification of Synchronous Circuits using Symbolic Logic Simulation and Temporal Logic," *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989, pp. 759–764.

[6] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, Vol. C-35, No. 8 (August, 1986), pp. 677–691.

[7] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler, "COSMOS: a Compiled Simulator for MOS Circuits," *24th Design Automation Conference*, 1987, 9–16.

[8] R. E. Bryant, "Boolean Analysis of MOS Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-6, No. 4 (July, 1987), 634–649.

[9] R. E. Bryant, and C.-J. H. Seger, "Formal Verification of Digital Circuits Using Symbolic Ternary System Models," *Computer-Aided Verification '90*, E. M. Clarke, and R. P. Kurshan, *eds.* American Mathematical Society, 1991, pp. 121–146.

[10] R. E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication," *IEEE Transactions on Computers*, Vol. 40, No. 2 (February, 1991), pp. 205–213.

[11] R. E. Bryant, "Formal Verification of Memory Circuits by Switch-Level Simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 10, No. 1 (January, 1991), pp. 94–102.

[12] R. E. Bryant, "A Methodology for Hardware Verification Based on Logic Simulation," *J.ACM*, Vol. 38, No. 2 (April, 1991), pp. 299–328.

[13] R. E. Bryant, D. E. Beatty, and C.-J. H. Seger, "Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation," *28th Design Automation Conference*, June, 1991, pp. 297–402.

[14] J. A. Brzozowski, and M. Yoeli. "On a Ternary Model of Gate Networks." *IEEE Transactions on Computers C-28*, 3 (March 1979), pp. 178–183.

[15] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," *27th Design Automation Conference*, 1990, pp. 46–51.

[16] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages*, Vol. 8, No. 2 (April, 1986), pp. 244–263.

[17] E. M. Clarke, O. Grumberg, and D. E. Long, "Model Checking and Abstraction," *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*, Jan., 1992.

[18] O. Coudert, C. Berthet, and J. C. Madre, "Verification of Sequential Machines using Boolean Functional Vectors," *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989, pp. 111–128.

[19] O. Coudert, J.-C. Madre, and C. Berthet, "Verifying temporal properties of sequential machines without building their state diagrams," *Computer-Aided Verification '90*, E. M. Clarke, and R. P. Kurshan, *eds.* American Mathematical Society, pp. 75–84.

[20] J. A. Darringer, "The Application of Program Verification Techniques to Hardware Verification," *16th Design Automation Conference*, 1979, pp. 375–381.

[21] S. Devadas, H.-K. T. Ma, and A. R. Newton, "On the Verification of Sequential Machines at Differing Levels of Abstraction," *24th Design Automation Conference*, 1987, pp. 271–276.

[22] T. Kam, and P. A. Subramanyam, "Comparing Layouts with HDL Models: A Formal Verification Technique," *International Conference on Computer Design*, IEEE, 1992, pp. 588–591.

[23] M. Gordon, R. Milner, and C. Wadsworth, "Edinburgh LCF", *Lecture Notes in Computer Science*, No. 78, Springer Verlag, 1979.

[24] M. Gordon, "Why higher-order logic is a good formalism for specifying and verifying hardware," *Formal Aspects of VLSI Design*, G. Milne and P. A. Subrahmanyam, *eds.*, North-Holland, 1986, pp. 153–177.

[25] Michael J. C. Gordon et al., *The HOL System Description*, Cambridge Research Centre, SRI International, Suite 23, Miller's Yard, Cambridge CB2 1RQ, England.

[26] M.J.C. Gordon and T.F. Melham (eds.), "Introduction to HOL", Cambridge University Press, 1993.

[27] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and Verifying Real-Time Systems by Means of the Synchronous Data-F low Language LUSTRE," *IEEE Transactions on Software Engineering*, Vol. 18, No. 9 (September, 1992), pp. 785–793.

[28] J. Joyce, "Generic Structures in the Formal Specification and Verification of Digital Circuits", *The Fusion of Hardware Design and Verification*, G. Milne, *ed.*, North Holland, 1988, pp. 50–74.

[29] J. Joyce and C. Seger, "Linking BDD-Based Symbolic Evaluation to Interactive Theorem-Proving", *30th Design Automation Conference*, 1993, pp. 469–474.

[30] J. S. Jephson, R. P. McQuarrie, and R. E. Vogelsberg, "A Three-Level Design Verification System," *IBM Systems Journal* Vol. 8, No. 3 (1969), pp. 178–188.

[31] R. P. Kurshan, and K. L. McMillan, "Analysis of Digital Circuits Through Symbolic Reduction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 10, No. 11 (November, 1991), pp. 1356–1371.

[32] C. A. Mead, and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[33] R. Milner, “A Proposal for Standard ML”, *Proceedings of ACM Conference on LISP and Functional Programming*, Austin, TX, Aug. 1984, pp. 184–197

[34] A. Pnueli, “The Temporal Logic of Programs,” *18th Symposium on the Foundations of Computer Science*, IEEE, 1977, pp. 46–56.

[35] D. S. Reeves, and M. J. Irwin, “Fast Methods for Switch-Level Verification of MOS Circuits”, *IEEE Transactions on CAD/IC*, Vol. CAD-6, No. 5 (Sept., 1987), pp. 766–779.

[36] C-J. Seger, and R. E. Bryant, “Modeling of Circuit Delays in Symbolic Simulation”, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989, pp. 625–639.

[37] C-J. Seger, and R. E. Bryant, “Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories”, *Formal Methods in System Design*, Vol. 6, March 1995, pp. 147–189, 1995.

[38] C. Seger and J. Joyce, “A Mathematically Precise Two-Level Formal Hardware Verification Methodology”, Technical Report 92-34, Department of Computer Science, University of British Columbia, December 1992.

[39] J. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[40] D. Turner, “MIRANDA: A Non Strict Functional Language with Polymorphic Types”, in *Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, Springer Verlag, Vol. 201, 1985.

[41] P. Wolper, “Expressing Interesting Properties of Programs in Propositional Temporal Logic, ” *Proc. 13th Annual ACM Symposium on Principles of Programming Languages*, Jan., 1986, pp. 184–193.

Index