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

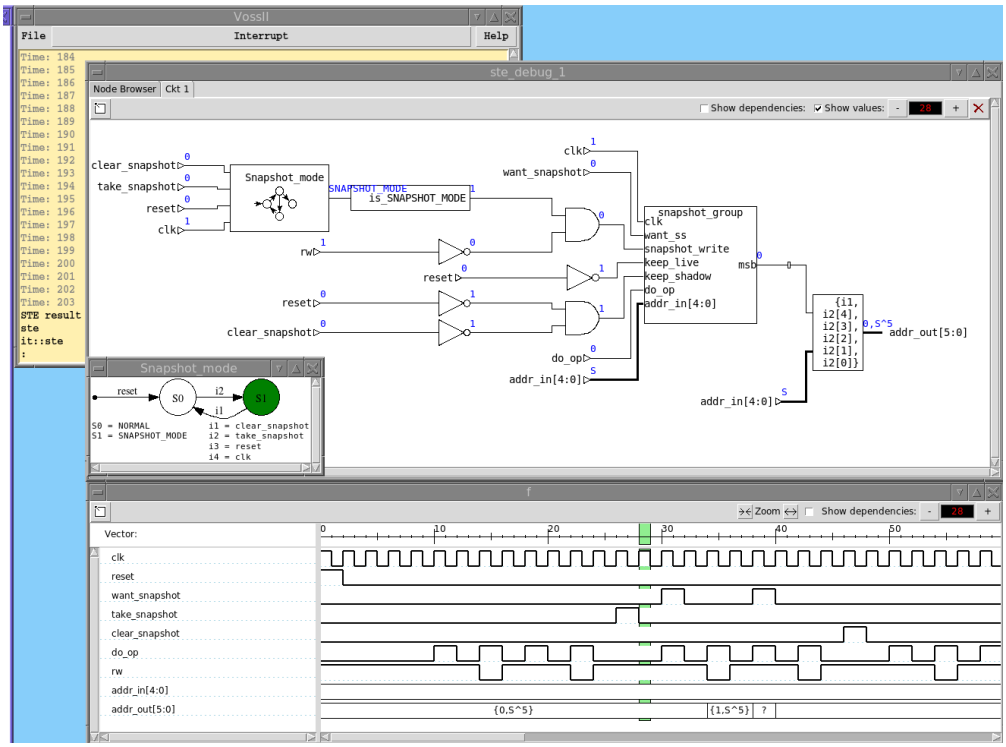
Carl-Johan H. Seger
Email: cjhseger@gmail.com

February 7, 2020

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

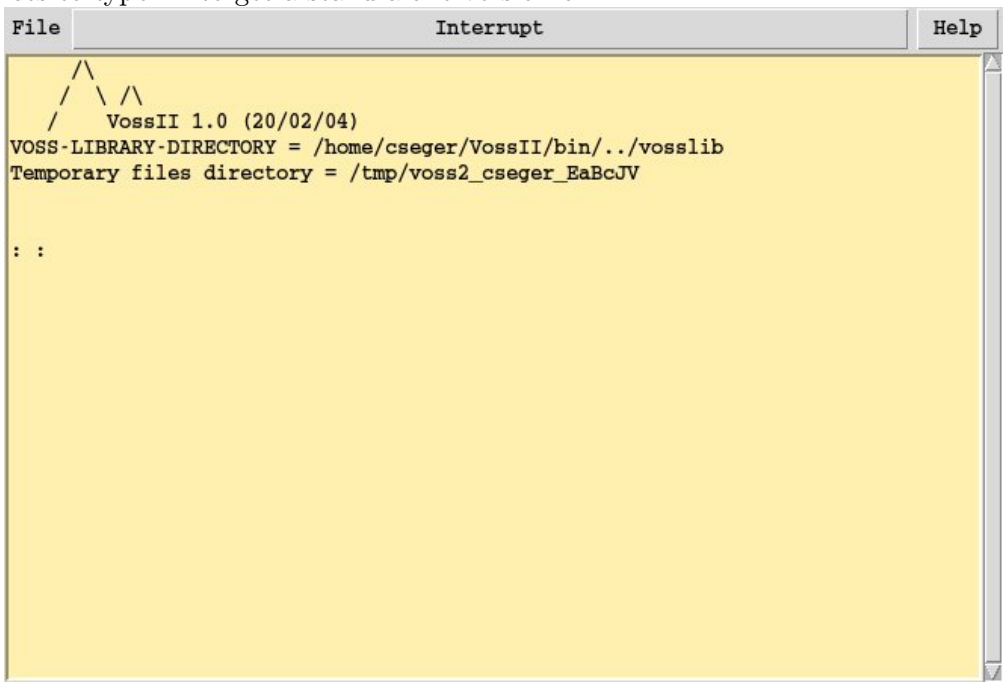
In this section we provide an introduction to the functional language fl.

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 the top-level one can for example: define functions (possibly of arity 0), define new concrete types, evaluate expressions, and modify the parser.

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.

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 continuously 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 **??** 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.

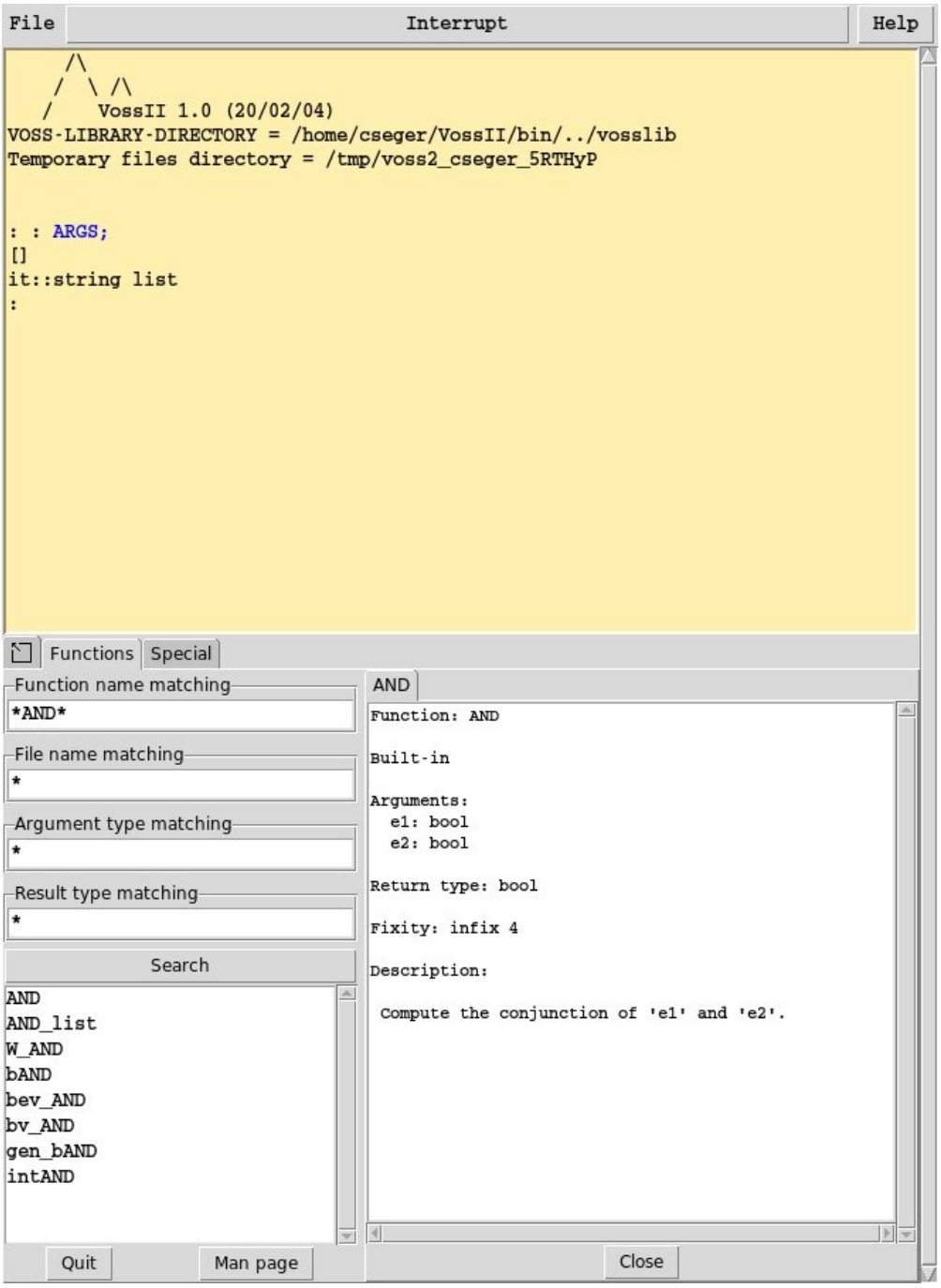
Any additional arguments to fl will be stored in the fl expression ARGS as a list of strings. For example:

```
% 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 right corner of the user interface. Whenever a function is defined, it is added to the online help system. Furthermore, if the function declaration is preceded 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 search 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:



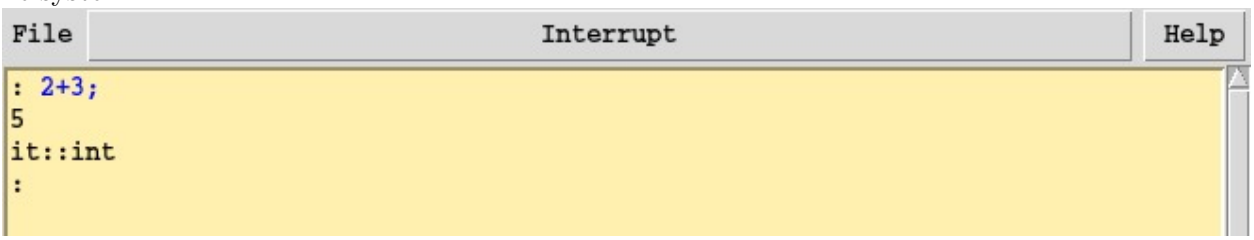
If the size of the help window is too small, and you don't want to make the whole fl window larger, you can click on the "detach" button beside the Functions tab to detach the help system.

1.3 Embedding in editors

A very convenient way of using **VossII** is to run it within an editor. There are currently two supported editors: vim and emacs. Both of these provide similar capabilities, but using different key-bindings to better follow the traditions of each editor. For details on how to enable vim or emacs integration, see the README files in `<VossII installation directory>/modes/vim` or `<VossII installation directory>/modes/emacs`.

1.4 Expressions

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



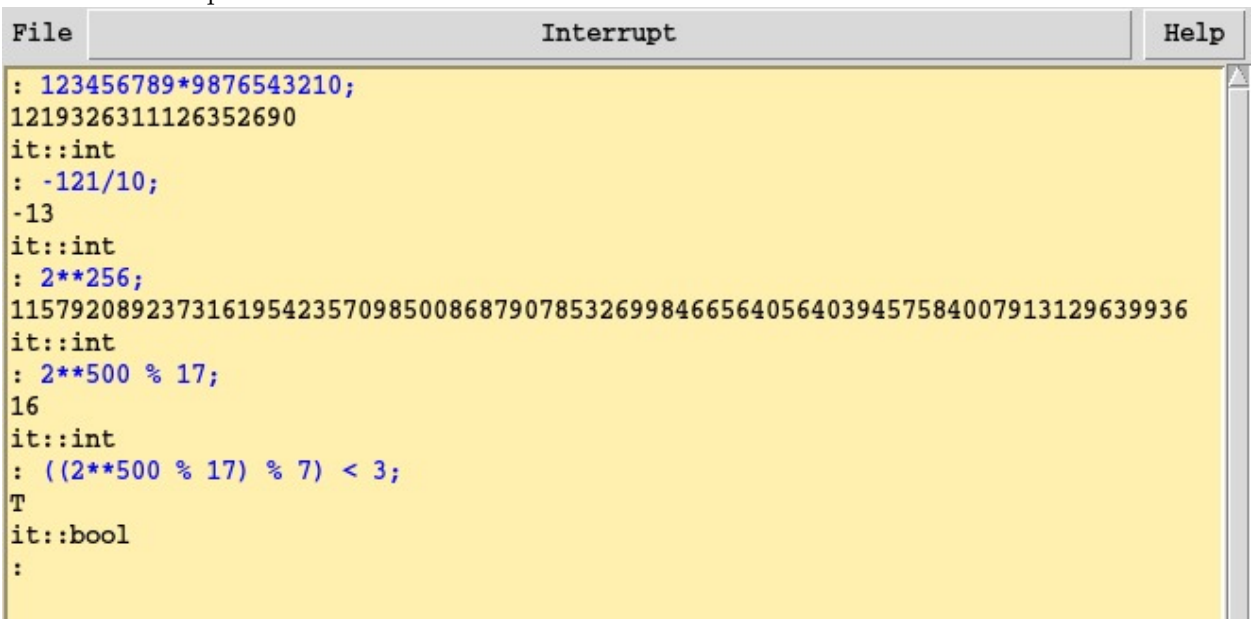
Here we simply evaluated the expression 2+3 and fl reduced it to normal form; in this case computed the result 5.

Note that **VossII** stores the result of the most recent expression in a variable called `it`. Thus, if you evaluate an expression, you can get a "handle" to it by using `it`.

Here we also illustrated the if-then-else construct used in fl.

1.5 Integer Expressions

VossII provides arbitrary precisions integers as default. All the usual arithmetic operations are available. For example:



It should be pointed out that all the arithmetic operations are overloaded with other data types (e.g., floating point numbers). As a result, the type inferred for an expression is sometimes more

general than expected. For example:

```
File                    Interrupt                    Help
: let double x = x+x;
double::[+] *->*
:
```

is not of type `int->int`, bit more general since it depends on the overloaded `+` function. Of course, you can use it on integers as you would expect:

```
File                    Interrupt                    Help
: double 12;
24
it::int
:
```

We will return to this later.

1.6 Floating Point Expressions

The `float` type represents double precision floating point numbers. A float must have a decimal point and a fraction digit. It can additionally have an exponent (e). For example:

```
File                    Interrupt                    Help
: 1.0;
1
it::float
: .34;
0.34
it::float
: -1.73e-12;
-1.73e-12
it::float
: 1.0 / 2.0;
0.5
it::float
: sin(pi*45.0/180.0);
0.707107
it::float
: round(-3.7);
-4
it::int
:
```

The following floating point operations are available at start:

```
+ - * /                      -- The normal arithmetic operations

< <= = != > >=             -- The normal relational operations

**                         -- Raised to the power

abs                        -- Absolute value
ceil                       -- Smallest integer >= value
floor                      -- Largest integer <= value
round                      -- Integer closest to value

sqrt                       -- Square root

sin cos tan                -- Sin, cos, and tangent in radians
asin acos atan             -- Arcsin, Arccos, etc. in radians

exp                        -- e to the power
log                        -- Natural logarithm
log10                      -- Logarithm based 10

float2str                  -- Convert float to a string
int2float                  -- Convert an integer to a string
str2float                  -- Convert a string to a float

pi                         -- The constant Pi.
```

1.7 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.

The constants true and false are denoted T and F respectively.

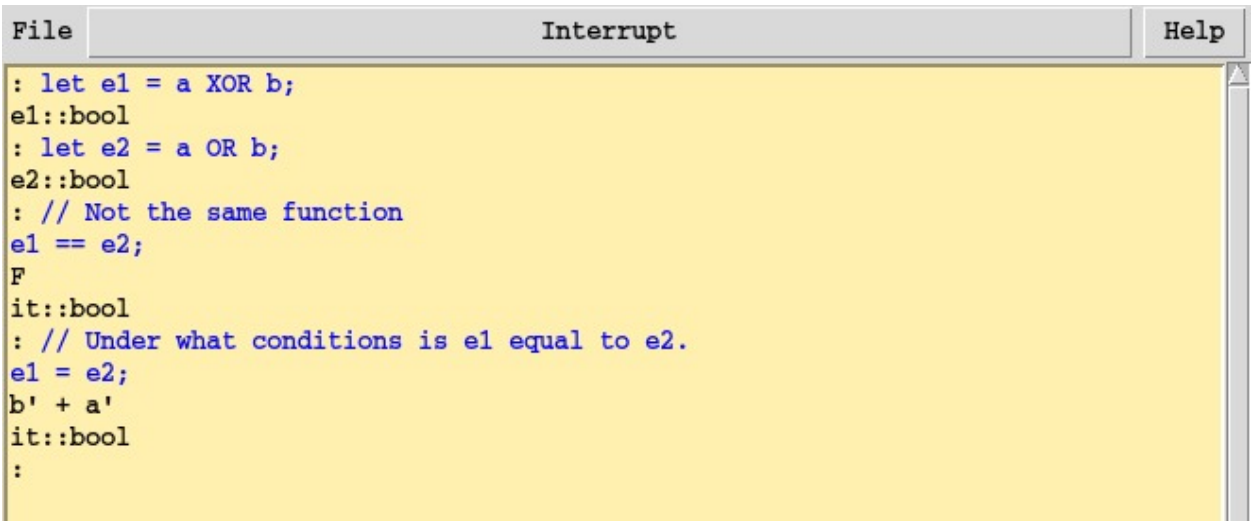
Boolean variables are created by variable s, where s is of type string. If several variables (including vectors) are needed, the function VARS is convenient since it both creates a zero-arity fl function for the bit or bit-list (depending on whether the name is a single node name or a vector name). For example:

```
File                    Interrupt                    Help
: variable "q";
q
it::bool
: VARS "a b c[3:0] d[1:0][2:0]";
a::bool
: b::bool
: c::bool list
: d::bool list
: : a;
a
it::bool
: b;
b
it::bool
: c;
[c[3],c[2],c[1],c[0]]
it::bool list
: d;
[d[1][2],d[1][1],d[1][0],d[0][2],d[0][1],d[0][0]]
it::bool list
:
```

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 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. To illustrate this, consider the following example:



It should be pointed out that the equal and not equal functions (`=` and `!=`) operate over any structure.

Note that unless an ordering has been installed explicitly (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 in the global BDD order and that they will occur in exactly this order.

Finally, by default, **VossII** will perform a dynamic re-ordering of the BDD variables when certain thresholds are exceeded. If this is not desirable, the vossrc variable "DYNAMIC-ORDERING" can be set to "NO". Similarly, a dynamic variable re-ordering can be triggered by the command `bdd_reorder` that takes an integer that determines the total number of sweeps that will be performed.

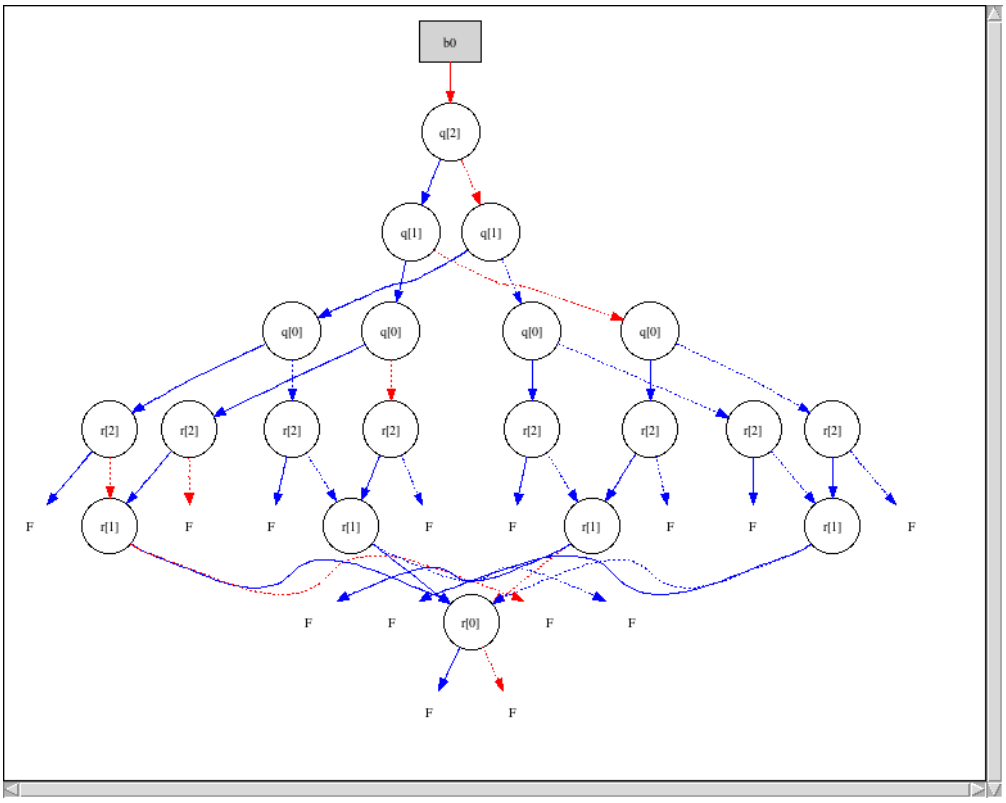
Note that during a dynamic variable re-ordering, a small status window will be opened at the bottom of the fl window showing the current status of the re-ordering as well as providing a button to abort the re-ordering.

1.7.1 Displaying and Analyzing BDDs

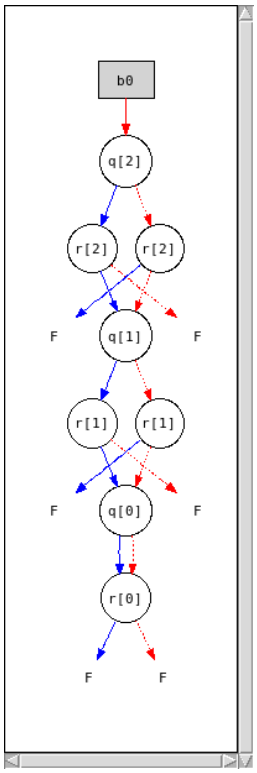
The default style for printing Boolean expressions is as a sum-of-products. Since this may require printing an extremely large expression, there is a user-settable limit on how many products that will be printed and the maximum size of a product. For more details how to modify these two parameters, see Section 1.27. For smaller BDDs, the function draw_bdds can also be very useful.

```
File                    Interrupt                    Help
: VARS "q[2:0] r[2:0]";
q::bool list
: r::bool list
: : // Force creation
q fseq r fseq ();
: // Order the variables sequentially
var_order ((md_expand_vector "q[2:0]")@(md_expand_vector "r[2:0]"));
["q[2:0]","r[2:0]","a","b","c[3:0]","d[1:0][2:0]","q","v"]
it::string list
: draw_bdds T [q = r];
: // Order the variables by interleaving them MSB to LSB
var_order (interleave [(md_expand_vector "q[3:0]"),
                       (md_expand_vector "r[3:0]")])
;
["q[3]","r[3]","q[2]","r[2]","q[1]","r[1]","q[0]","r[0]","a","b","c[3:0]","d[1:0
][2:0]","q","v"]
it::string list
: draw_bdds T [q = r];
:
```

which would create the display



and



For larger BDDs, there are a number of helpful functions that can be used to understand the BDD. The are:

```
depends
forcing
truth_cover
example
enumerate_examples
```

Depends is used to return a list of all BDD variables an expression depends on. Note that the expression can be of any type since depends will recursively traverse the whole expression looking for BDDs.

Forcing f returns a substitution list (list of (variable,value) pairs) for all variables that must take on some specific value for f to be true.

Truth_cover takes a list of variables (vars) and a boolean function (f) and returns the number of assignments to the variables in vars that makes f true.

Example takes a preference (true or false) and a BDD f and returns a substitution list that makes f true. If there is a choice, the preference decides whether T or F should be selected.

Finally, `enumerate_examples` is a convenient way of getting some randomly selected scalar values that makes a BDD true.

1.7.2 Symbolic If-then-else

One of the more unusual features of fl is that conditionals can be symbolic. For example:

```
File                    Interrupt                    Help
: let e1 = a XOR b;
e1::bool
: // If-then-else with symbolic condition.
e1 => a | b;
a
it::bool
:
```

There is some restrictions on how these types of conditional if-then-else's can be used. With a few exceptions, both the then-case and the else-case has to be structurally the same. Effectively, they can only differ in Boolean values. Thus

```
File                    Interrupt                    Help
: a => (1,[b]) | (1,[a]);
(1, [a&b])
it::int#(bool list)
:
```

is acceptable, but

```
File                    Interrupt                    Help
: a => (1,[b]) | (2,[a]);
Failure:    if-then-else over non-matching structures
it::int#(bool list)
:
```

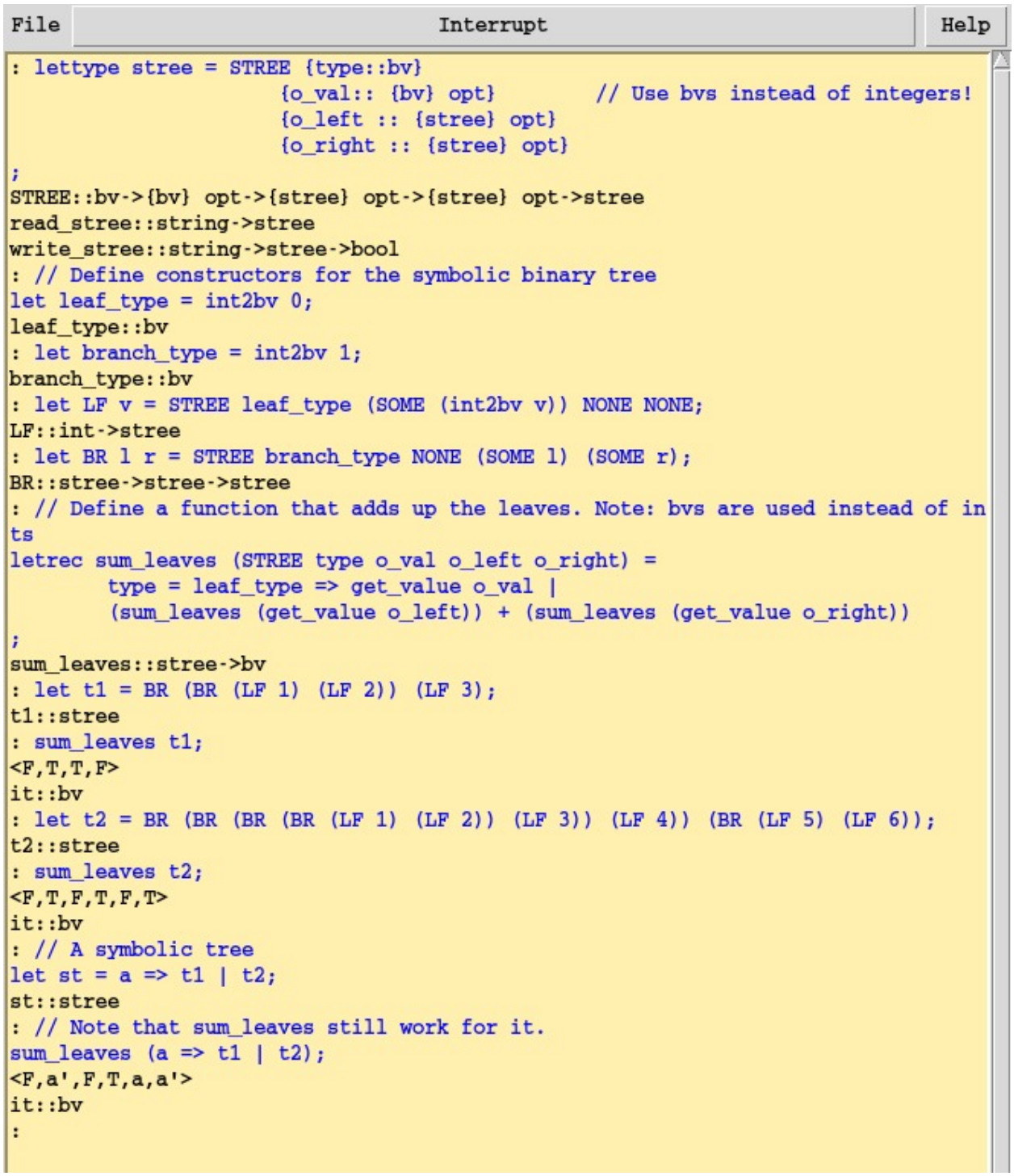
is not.

There are two major exceptions to this. First, bvs (more about these in Section 1.32) automatically adjust to make the structures equal. For example:

```
File                    Interrupt                    Help
: let v1 = int2bv 3;
v1::bv
: let v2 = int2bv 12;
v2::bv
: let res = a => v1 | v2;
res::bv
: res;
<F,a',a',a,a>
it::bv
:
```

Note that res effectively encodes symbolically both of the alternatives.

The second exception to the requirement of structural similarity is how the constructors for the option type is handled. To make it possible to create symbolic data structures, the handling of the optional constructors NONE and SOME is enhanced. In particular, the optional data type constructor NONE matches not only NONE but more importantly the SOME constructor and the result is the SOME value. An example of using this, consider the following symbolic binary tree structure.



1.7.3 Advanced BDD operations

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 list of variables. 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 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
:
```

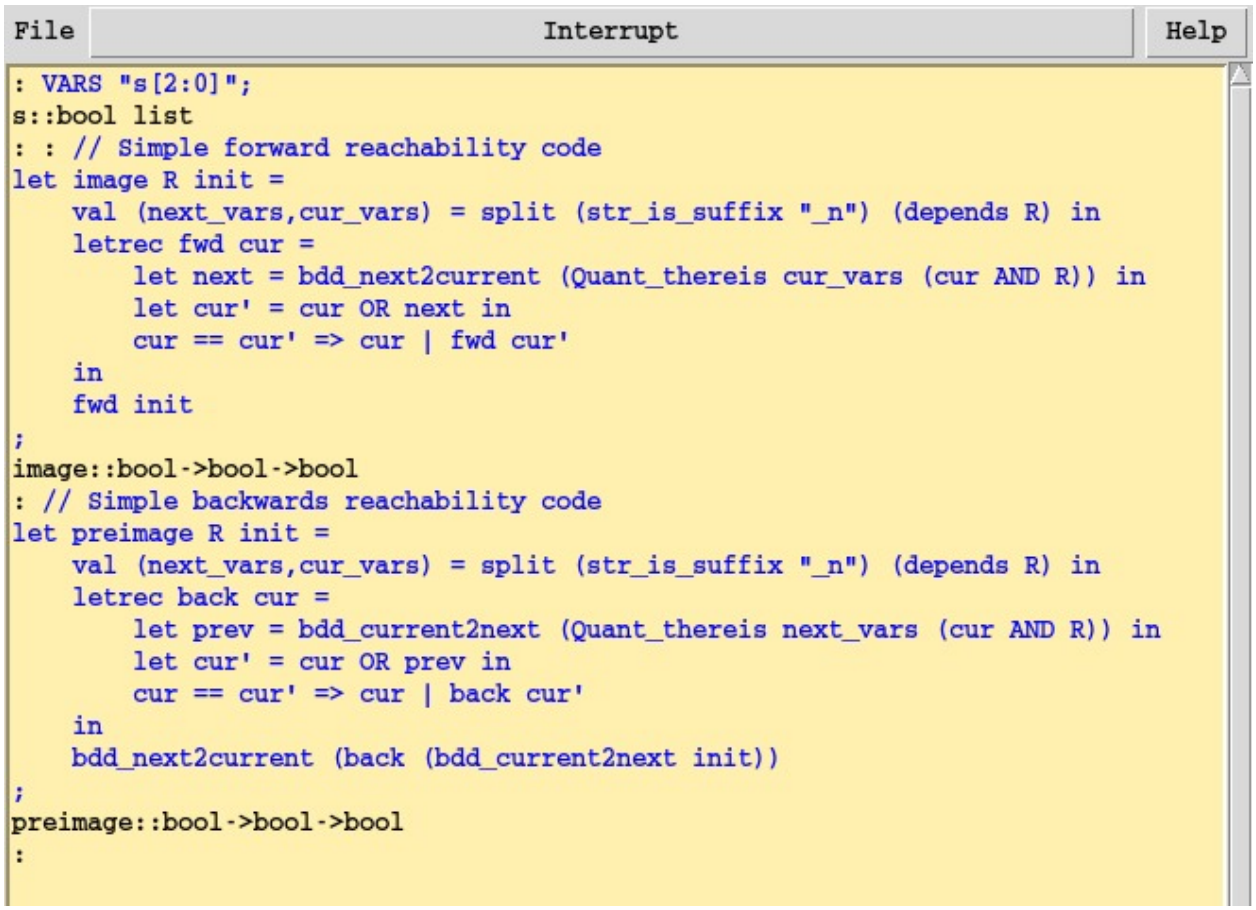
A common operation when implementing a symbolic model checker, is to perform a quantification for a product of two BDDs. Rather than doing the conjunction and then the quantification, the functions `relprod_forall` and `relprod_thereis` perform perform this combinations of operations much more efficiently. See the help system for more details.

A common operation with BDDs is to simultaneously substitute some expressions for (some) variables in the BDD. For this purpose, there is a function called `substitute`. This function takes a substitution list (list of string name–Boolean expression pairs), and an arbitrary fl expression. It then traverses the expression and performs the substitution on every object of type bool it encounters.

File Interrupt Help

```
: VARS "a b c d";
a::bool
: b::bool
: c::bool
: d::bool
: : let e = [(1,[a AND b XOR c]), (2,[T,T,a,b, d])];
e::int#(bool list) list
: e;
[(1, [a&b&c' + b'&c + a'&c]),(2, [T,T,a,b,d])]
it::int#(bool list) list
: substitute [("a",T),("b",F),("c",F),("d",F)] e;
[(1, [F]),(2, [T,T,T,F,F])]
it::int#(bool list) list
: substitute [("a",b),("b",a)] e;
[(1, [a&b&c' + b'&c + a'&c]),(2, [T,T,b,a,d])]
it::int#(bool list) list
:
```

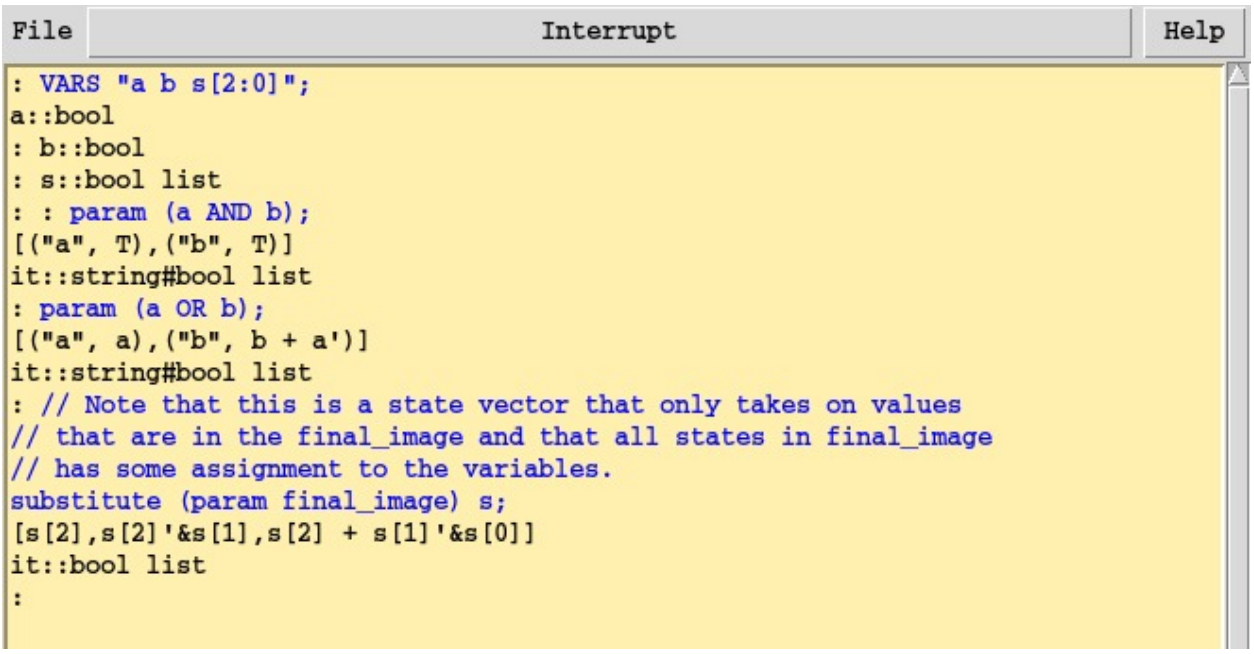
When a BDD variable 'x' is created, a "next-state" variable 'x_n' is also created. These two variables are always kept together in the variable order. As a result, the size of a BDD in terms of "current-state" variables is the same as it would in in terms of "next-state" variables. To make this conversion easier, there are two helpful functions: `bdd_current2next` and `bdd_next2current`. For example,



With these functions we could now do:

```
File                          Interrupt                          Help
: // Example of simple state machine
let transition_relation =
    let s' = bdd_current2next s in
    let mk_trans (f,t) = (s = (int2bl 3 f)) AND (s' = (int2bl 3 t)) in
    let my_fsm = [(0,1),
                  (1,0), (1,1), (1,2),
                  (2,5),
                  (3,4), (3,2),
                  (4,3),
                  (5,5)]
    in
    itlist (\tr.\r. mk_trans tr OR r) my_fsm F
;
transition_relation::bool
: // Compute the set of states reachable from init
let init = (s = int2bl 3 0);
init::bool
: let final_image = image transition_relation init;
final_image::bool
: enumerate_examples 20 (depends s) final_image;

s[2:0]=000
s[2:0]=001
s[2:0]=010
s[2:0]=101
: // Compute the set of states that could have taken us to final.
let final = (s = int2bl 3 3);
final::bool
: let res = preimage transition_relation final;
res::bool
: enumerate_examples 20 (depends s) res;

s[2:0]=011
s[2:0]=100
:
```

A very common problem is to create a vector of Boolean variables but restrict the possible values to be in a set. For example, one might want to have a vector of 4 bits that has the property that exactly one signal is high. For this problem the param is available. Param takes a boolean condition c and returns a substitution list that provides expressions for every variable in c such that 1) every truth assignment to the parametric variables will make c satisfied, and 2) for every substitution to the variables in c that makes c satisfied there is some truth assignment to the parametric variables so (param c) is that substitution.



If one wants to solve for only some variables, i.e., treat the remainder as symbolic constants, there is a fparam function available. See the help system for more details.

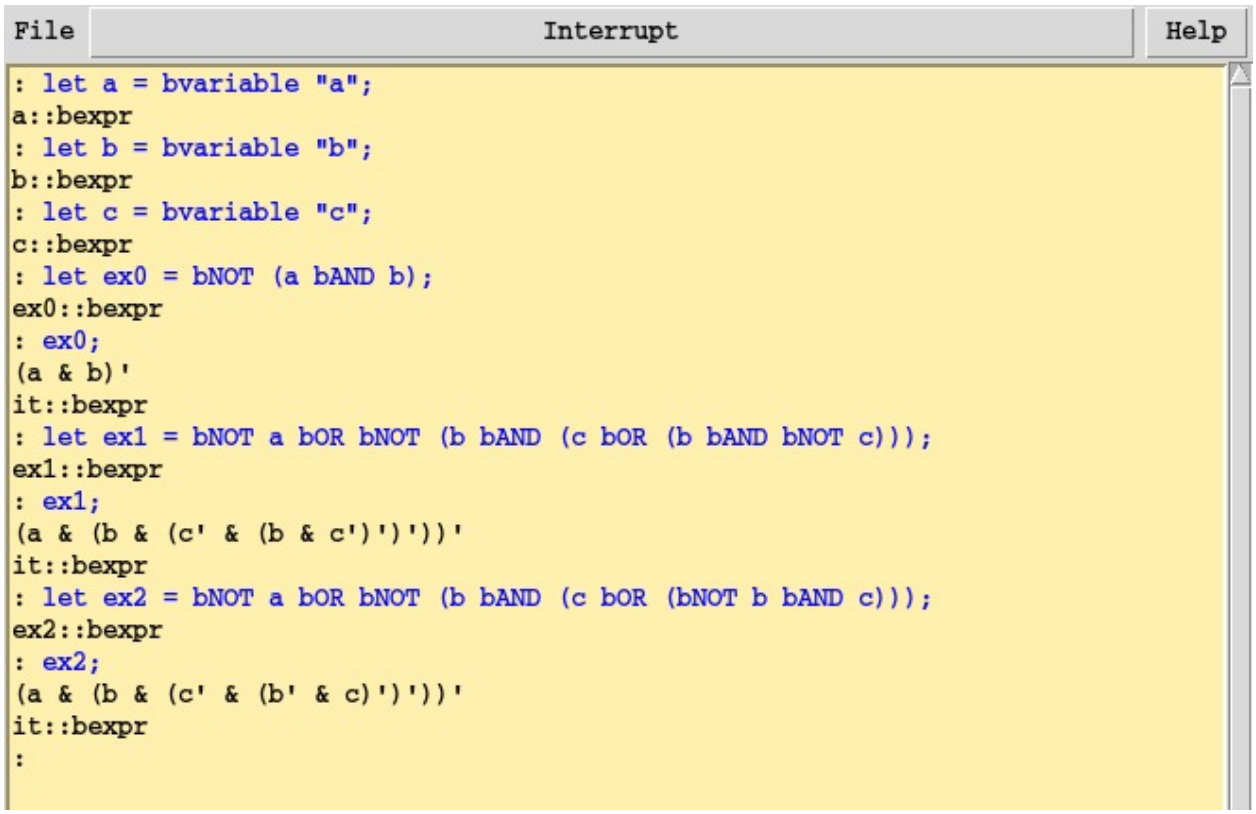
1.7.4 Structural BDD operations

Sometimes one needs to write a function that walks over a BDD and perform some operation. For this task, the functions top_cofactor and bdd_signature are useful. Of course, it is almost certain that any function walking over a BDD needs to be memoized since the BDDs are DAGs and thus a tree structured traversal can take time that grows exponentially with the size of the BDD!

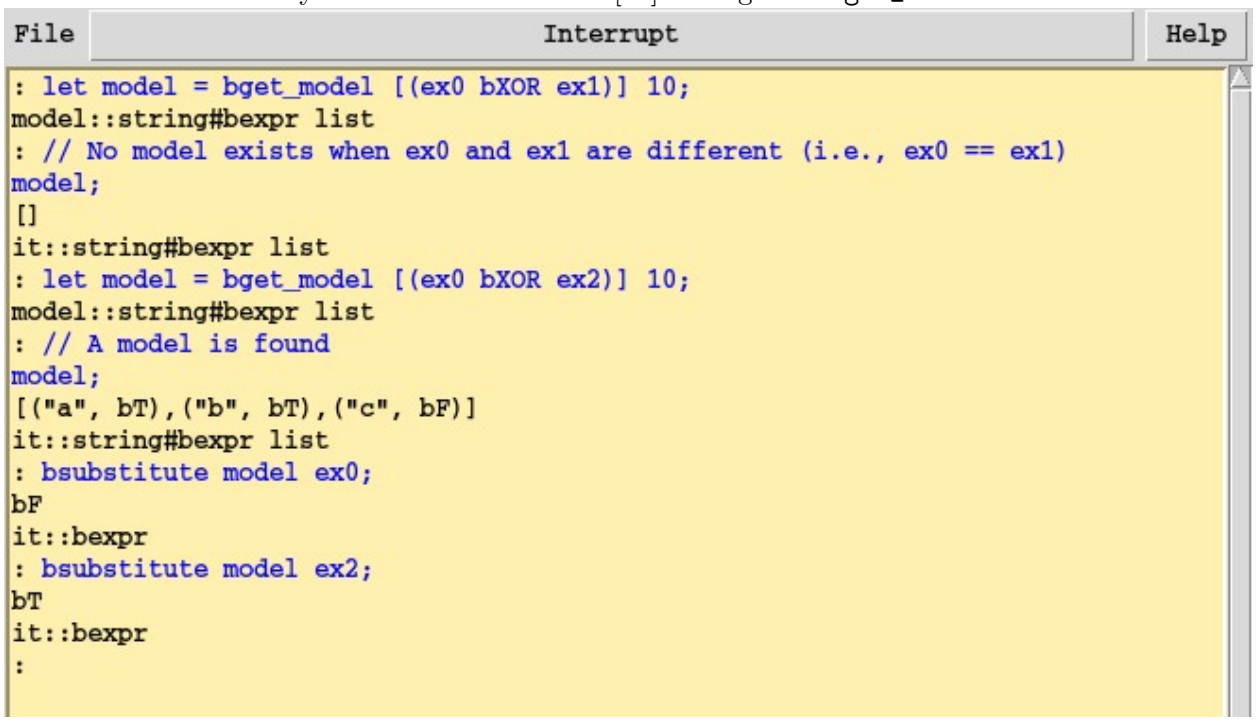
Finally, it is sometimes useful to save a collection of BDDs to disk to work with them in a fresh **VossII** invocation. Thus, bdd_save and bdd_load come in handy.

1.8 bexpr and SAT

If you don't want to use BDDs for representing Boolean expressions, fl provides `bexpr`, which are essentially AND-INVERT directed acyclic graphs. Most of the Boolean operations, have a bexpr equivalent that is named the same, except prefixed with a b. For example:



Since bexprs are not canonical, one must use a SAT solver to determine whether a bexpr is satisfiable or not. Currently **VossII** uses MiniSAT [41] through the `bget_model` function call.



Like with BDDs, there are functions to recursively walk over bexprs. Here the relevant functions are: bget_type, bget_arguments and bget_variable.

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 Printf Family of Functions

To make the conversion of values to strings easier, there is a family of functions called `sprintf` that takes a format string to guide the conversion. The type of the sprintf function is determined by the format string

The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %, and ends with a conversion specifier. In between there may be (in this order) zero or more flags, and an optional length modifier.

The arguments must correspond properly (after type promotion) with the conversion specifier. The arguments are used in the order given, where each '*' and each conversion specifier asks for the next argument.

The character % should be followed by zero or more of the following flags:

0 The value should be zero padded.

- The value should be left justified.

After this, one can give an optional decimal digit string (with nonzero first digit) specifying a field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given). Instead of a decimal digit string one may write `*` to specify that the field width is given in the next argument. A negative field width is taken as a '-' flag followed by a positive field width. If the resulting value does not fit in the width given, an exception is raised.

Finally, there is a conversion specifier, which is one of

b the integer argument will be converted to a binary number

o the integer argument will be converted to an octal number

d the integer argument will be converted to a signed decimal number

x the integer argument will be converted to a hexadecimal number

s the string argument will be copied out

S the list of strings argument will be copied out enclosed in [] and each string separated by a comma.

B the BDD (bool) argument will be converted to a string and copied to the output

For example:

```
File                    Interrupt                    Help
: let conv i = sprintf "%0*b is the same as %x or %d\n" 16 i i i;
conv::int->string
: conv 2;
"0000000000000010 is the same as 2 or 2
"
it::string
: conv 142;
"0000000010001110 is the same as 8e or 142
"
it::string
: conv (-142);
"1111111101110010 is the same as -8e or -142
"
it::string
:
```

There are also an fprintf, printf, and eprintf versions for writing the resulting string to a stream, stdout, or as an error message respectively.

Similarly, there is a `sscanf` family of functions that can be used to parse simple strings to values. It uses the same type of format string, except neither 'S' or 'B' are accepted. For example:

```
File                    Interrupt                    Help
: sscanf "(%d,%04b)-->%x" "(12,1101)-->4f";
(12, (13, 79))
it::int#(int#int)
:
```

1.11 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:

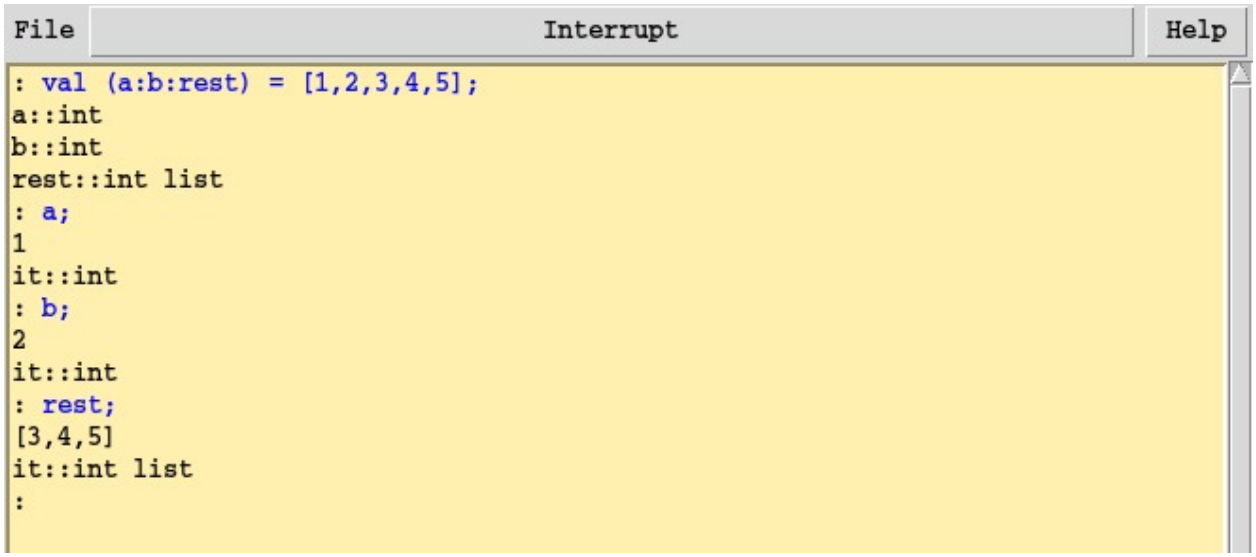
```
File                    Interrupt                    Help
: let x = 2;
x::int
: let y = x*5;
y::int
: let x = 12;
x::int
:
```

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
:
```

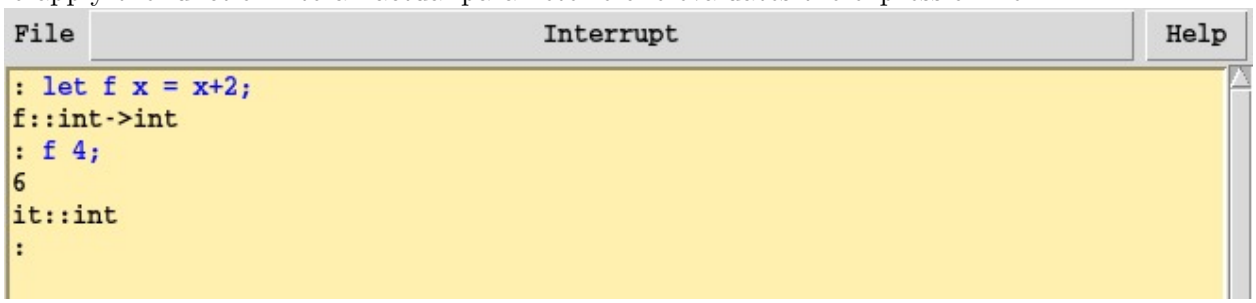
or



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 **??**.

1.12 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::[-(2)] (*#**)->(*#**)->*#**
: 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:

```
: let manhat_dist (x1,y1) (x2,y2) = (x2-x1, y2-y1);
manhat_dist::[-(2)] (*#**)->(*#**)->*#**
: (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 **??**.

1.13 Recursive Functions

The following is an attempt to define the factorial function:

```
: let fact n = n=0 => 1 | n*fact (n-1);
===Type error around line 60
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:

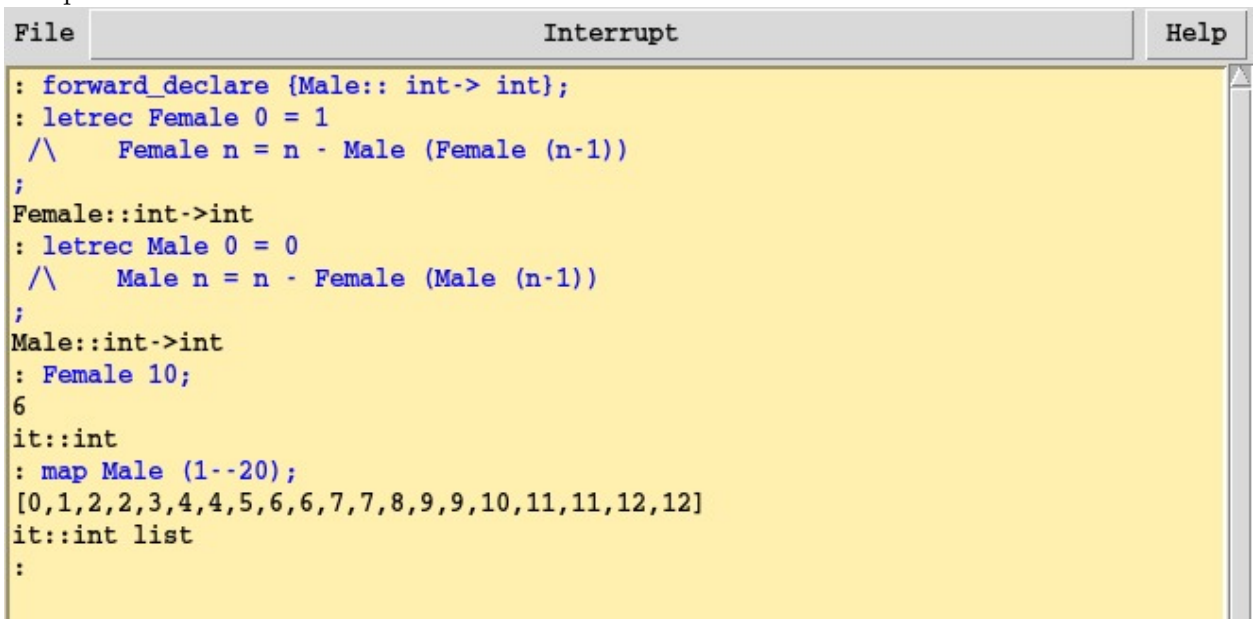
```
: let f n = n+1;
f::int->int
: let f n = n=0 => 1 | n*f (n-1);
f::int->int
: f 3;
9
it::int
:
```

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
:
```

1.14 Mutually Recursive Functions

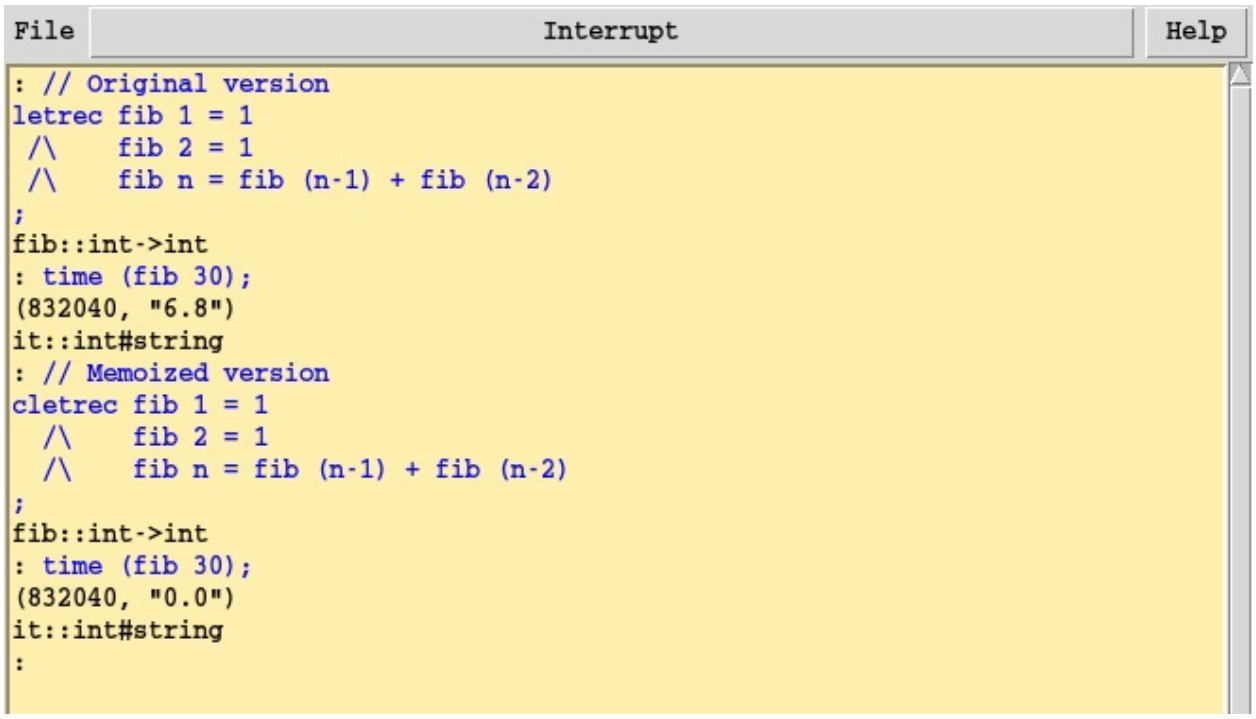
There is currently no direct support for mutually recursive functions, i.e., declaring a function f that needs to evaluate g, but g needs to evaluate f etc. The work-around for this problem is to forward declare one of the functions. The command: forward_declare allows a "stub" to be declared. When the declarations of this functions comes later, the stub is replaced with the actual definition. To illustrate this, consider defining functions that compute members of the Hofstadter Female and Male sequences:



It should be pointed out that a function that is to be forward declared, must be given the same type as it will get once it is defined.

1.15 Memoization

Sometimes, there are pure functions that take relatively simple inputs and are heavily used with a relatively small set of inputs and that take a long time to compute their results. For these types of functions, memoization [32] is often very useful. In fl, a function can be memoized simply by using `clet` or `cletrec` instead of `let` and `letrec` respectively. The classical example is Fibonacci number:

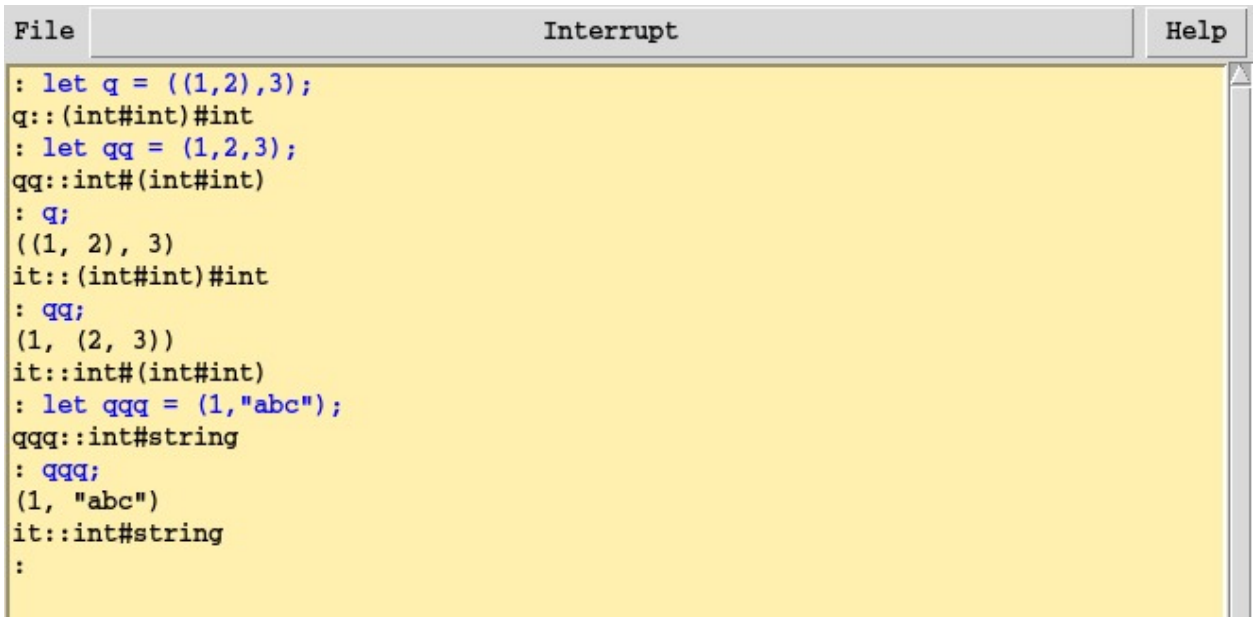


It is important to remember that a memoized function might not be evaluated every time it is called. Thus if the functions have side effects (e.g., prints out something), the memoization of the function will lead to unexpected results. Sometimes all subsequent calls will be found in the memo table. Sometimes only some of the subsequent calls will be found. Similarly, if the function depends on reference variables (i.e., the function is not pure but depends on state holding variables), the result will likely be completely incorrect. However, if the function is pure, memoizing it will never change the result.

It should be pointed out that one should not memoize any function. Not only can it increase the memory foot print significantly, for some functions it will slow down the program. In particular, when the arguments can be very large, and thus the hashing function take long time, and the computation relatively small, memoization is a bad idea.

1.16 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 $t\_1\#t\_2\#\ldots\#t\_n$. The standard functions on tuples are fst (first), snd (second), and the infix operation , (pair).



1.17 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
:
```

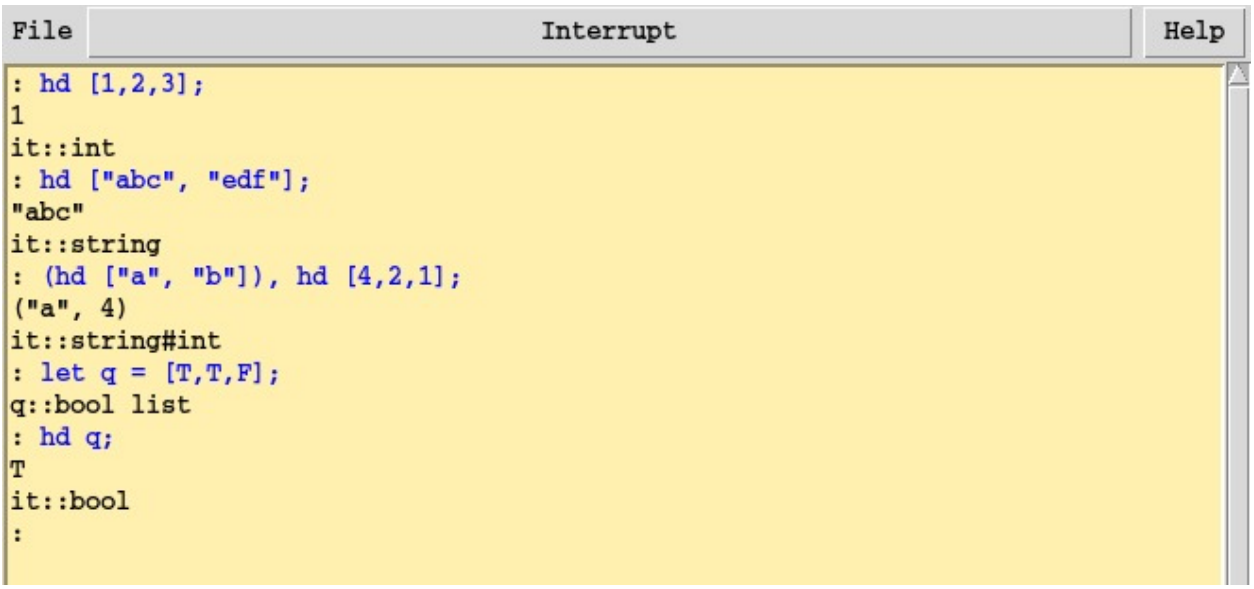
There are a large number of list functions built into fl. For example:

```
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
:
```

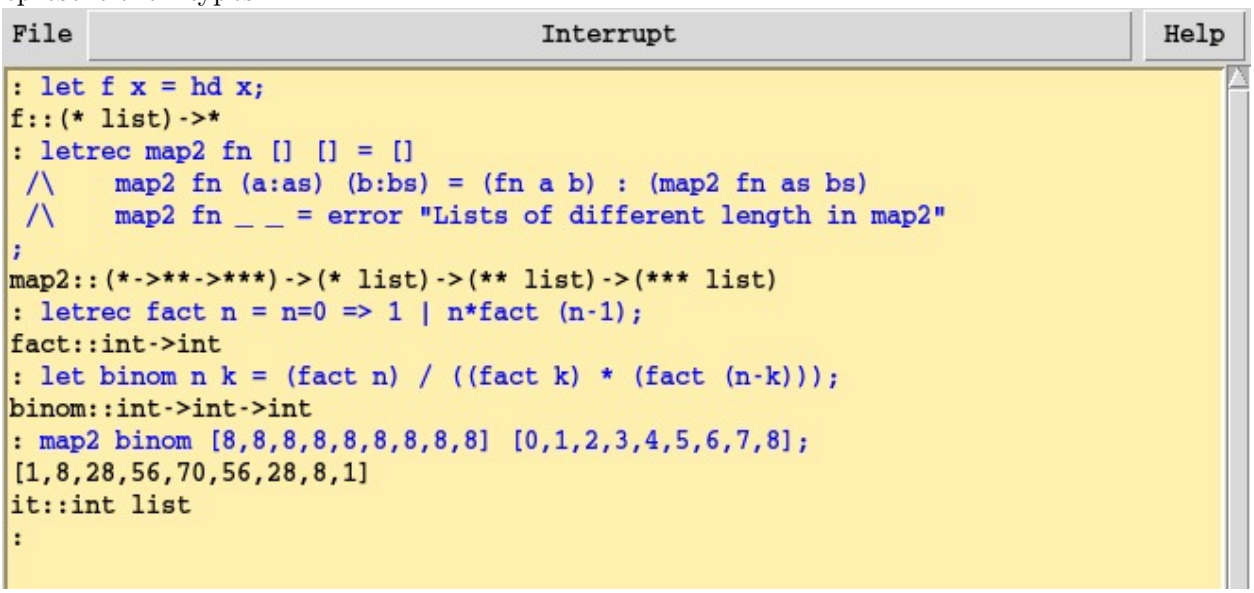
To find all such functions, use the help system and search for functions whose argument type matches "*list*".

1.18 Polymorphism

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



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.



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.

```
File                    Interrupt                    Help
: 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.

```
File                    Interrupt                    Help
: 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.

```
File                    Interrupt                    Help
: 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: string and int
===Type error around line 83
Inferred type is:
        int list
but its usage requires it to be of type:
        string list
:
```

In this respect, fl is similar to the functional language called Miranda[1] [42].

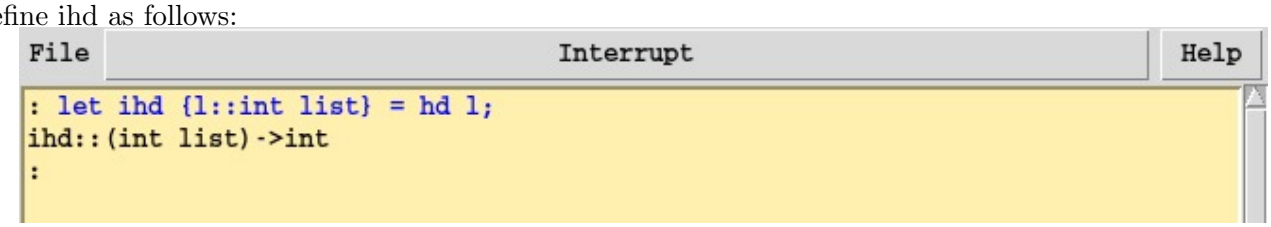
1.19 Type Annotations

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.

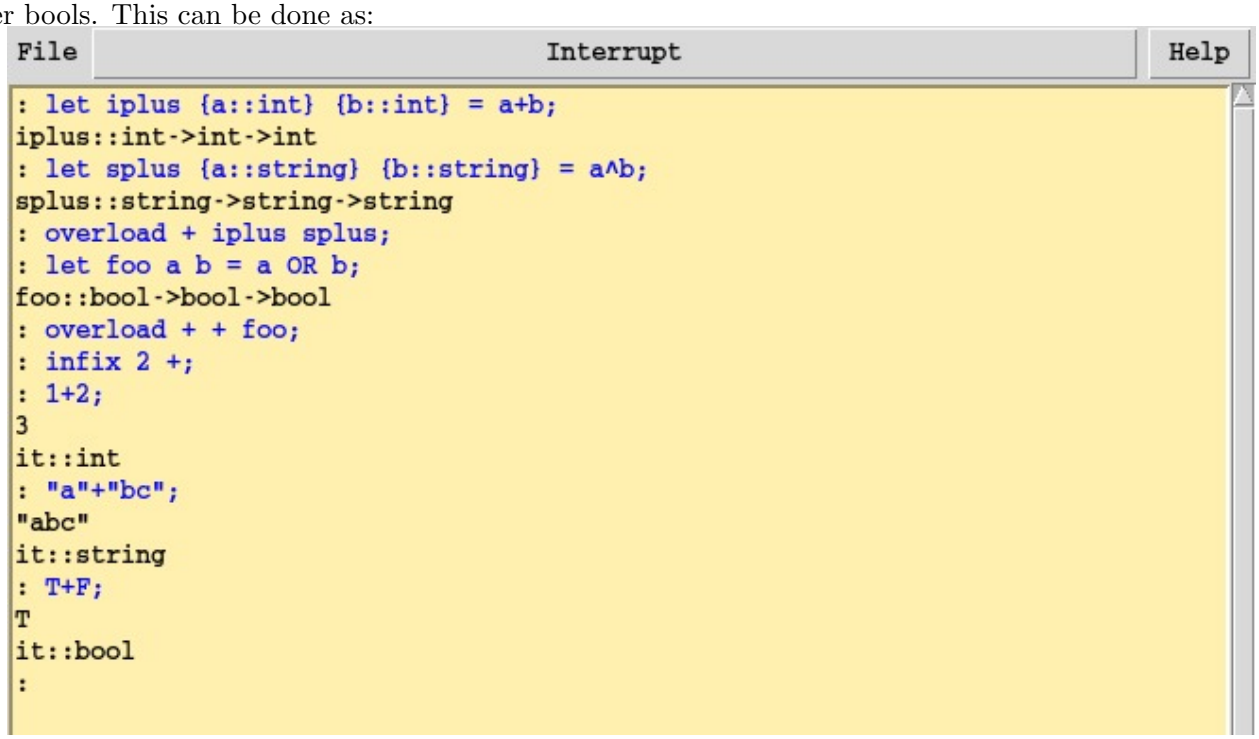
[1] Miranda is a trademark of Research Software Ltd.

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.

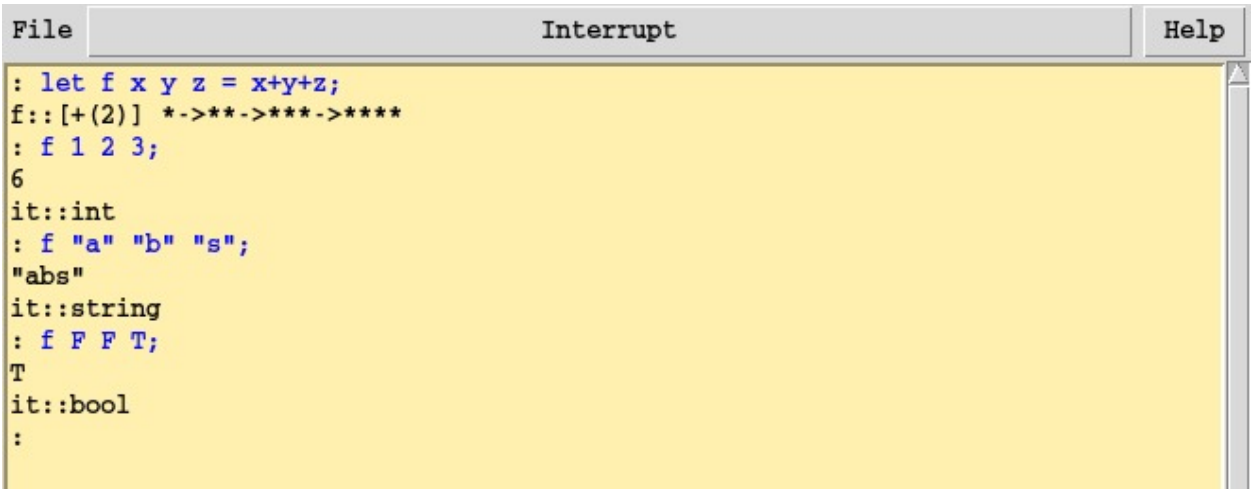
In fl a variable or expression can be annotated with its expected type by enclosing it in curly braces and decorate the expression with a type expression. For example, if we would like to define a function ihd that return the head of a list, but that only can be used on integer lists, we could define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



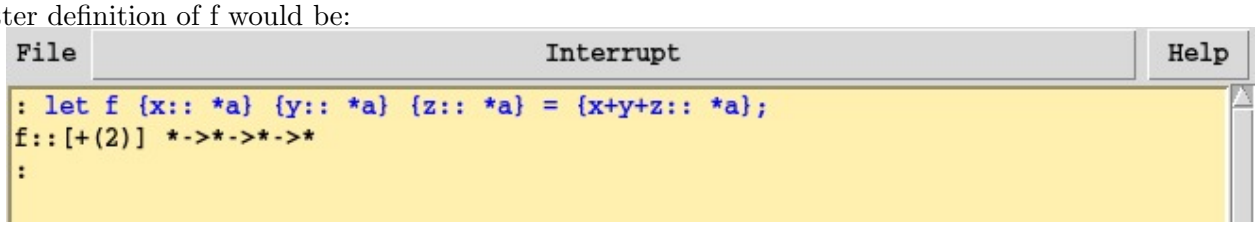
A more subtle example is the following. Assume we have overloaded the operator + to either operate over strings or integers. We then want to extend this overloading with yet another function over bools. This can be done as:



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



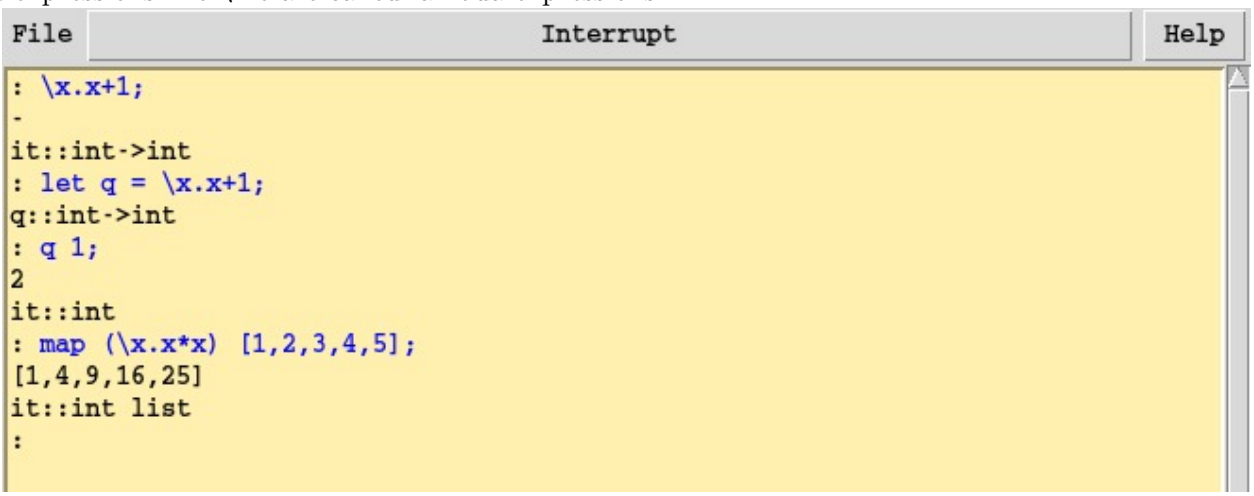
However, note that the type inferred for f is more general than one might like. After all, all the arguments to f as well as its return type, must have the same type. To capture this, a slightly better definition of f would be:



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!

1.20 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.



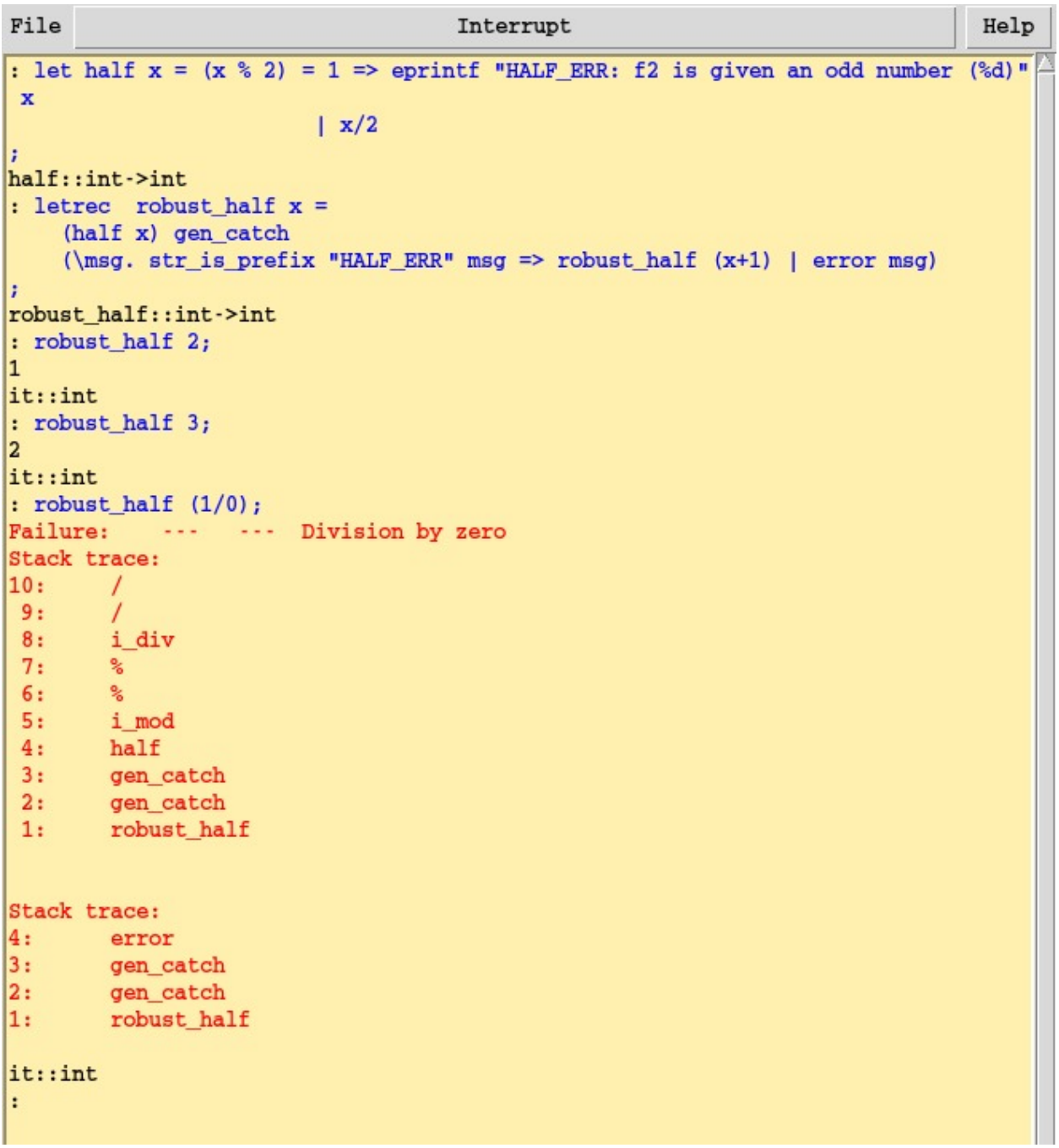
1.21 Failures

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:
2:      hd
1:      hd

it::int
: 1/0;
Failure:    ---  Division by zero
Stack trace:
3:      /
2:      /
1:      i_div

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

Stack trace:
1:      error

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:

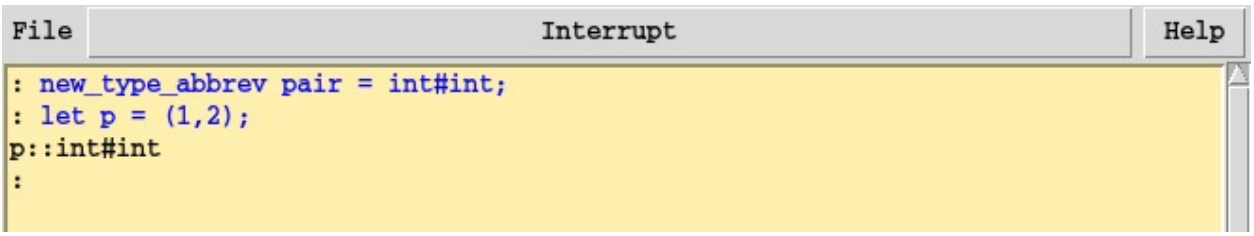


Here we catch only errors with "HALF_ERR" in the error message.

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.

1.22 Type Abbreviations

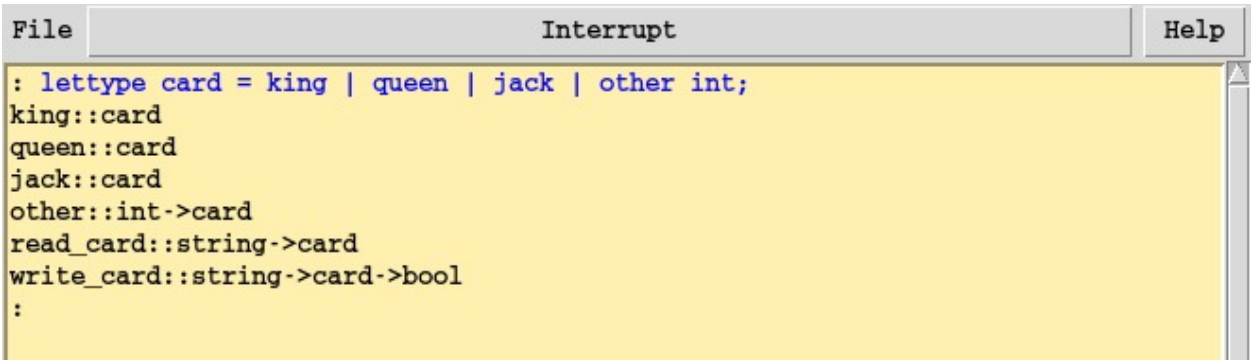
Types can be given names:



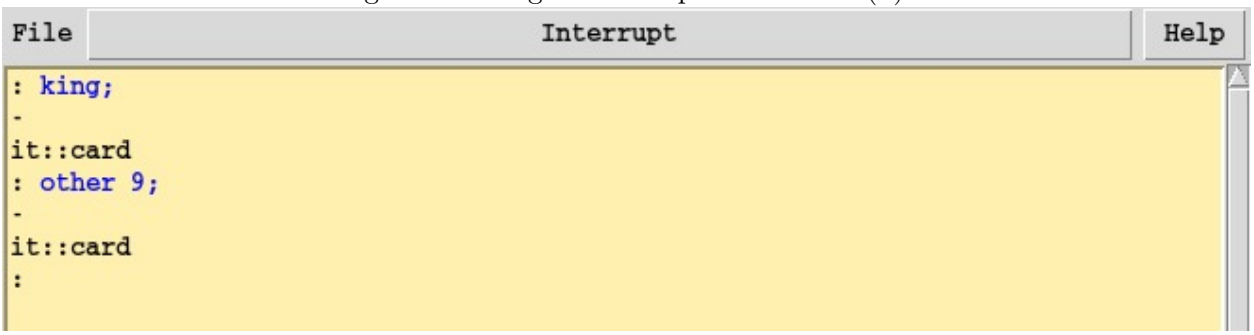
However, as can be seen from the example, the system does not make any distinction between the new type name and the actual type. It is purely a short hand that is useful when defining concrete types below.

1.23 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:



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).



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:

```
File                    Interrupt                    Help
: 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

```
File                    Interrupt                    Help
: 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:

```
File                    Interrupt                    Help
: 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:

```
File                          Interrupt                          Help
: 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:

```
File                          Interrupt                          Help
: 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, although it requires one to declare the type of the forward function(s).

1.24 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.

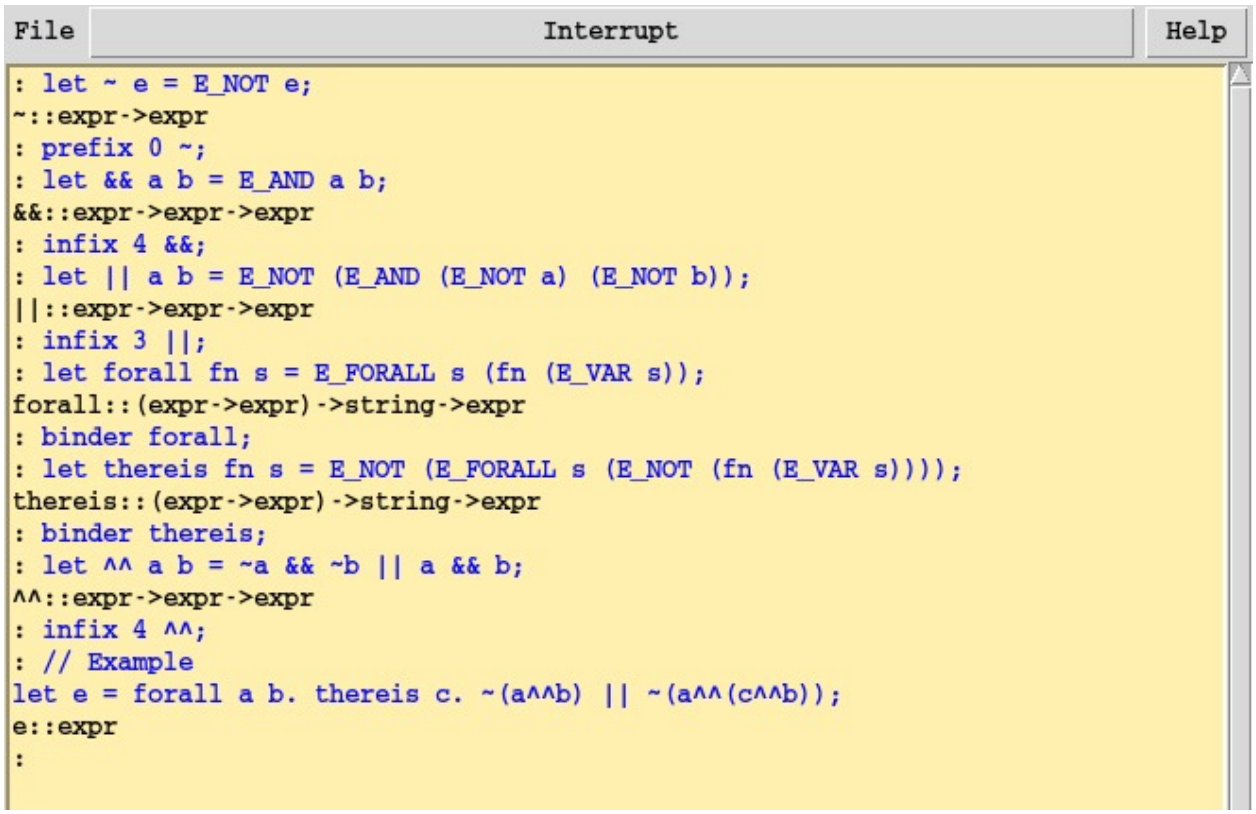
```
File                          Interrupt                          Help
: lettype expr =  E_FORALL string expr |
                  E_VAR string |
                  E_TRUE |
                  E_AND expr expr |
                  E_NOT expr
;
E_FORALL::string->expr->expr
E_VAR::string->expr
E_TRUE::expr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

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

File | Interrupt | Help

```
: 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 convenient to input expressions, one usually defines operators and give them suitable fixities.

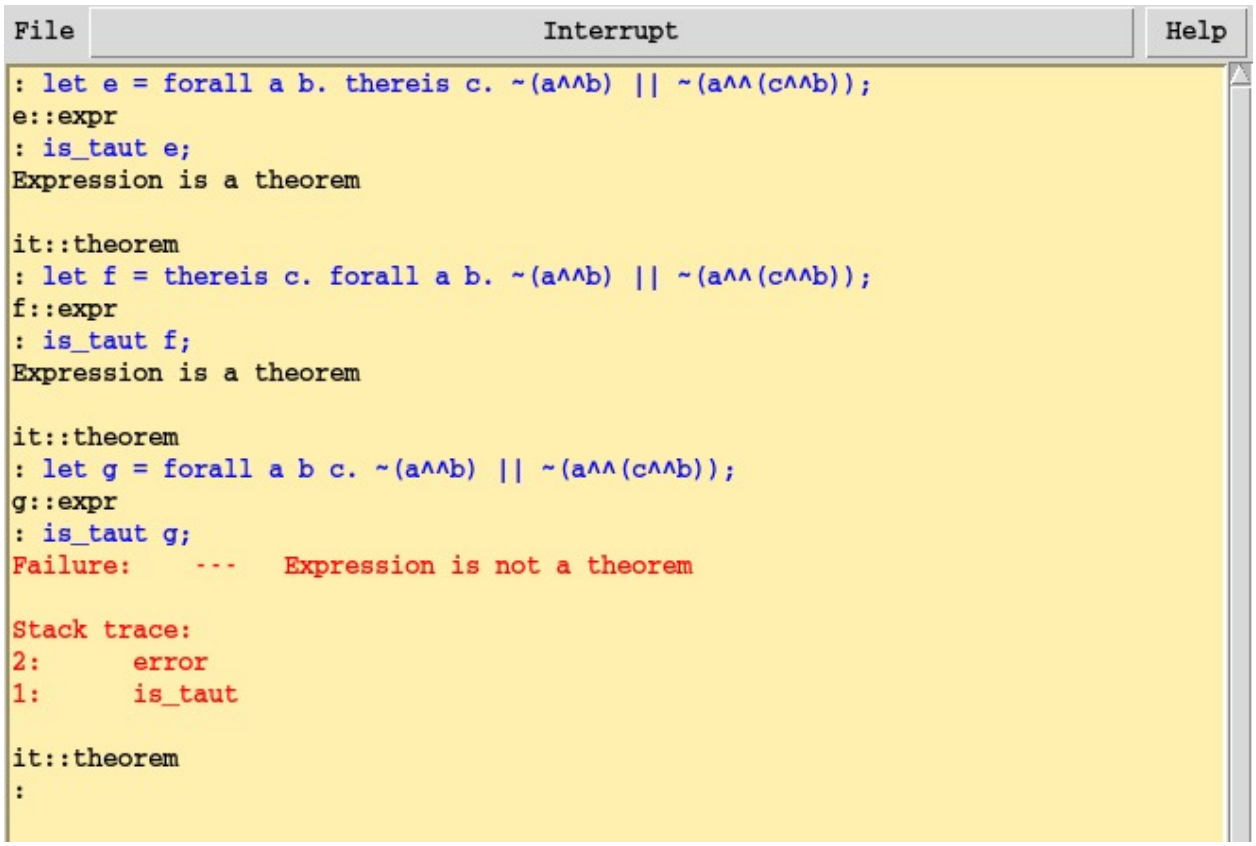


We then define the concrete type theorem and the constructor function is_taut. Note that we also define a couple of help functions. However, only the is_taut function is exported out of the abstract type, and thus is the only way of creating a theorem.

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;
THM::expr->theorem
read_theorem::string->theorem
write_theorem::string->theorem->bool
: let Ptheorem t =
    let PP (THM e) = fprintf stdout "Expression is a theorem\n" in
    (PP t) fseq ""
;
Ptheorem::theorem->string
: install_print_function Ptheorem;

: let is_taut e =
    (expr2bool e == T) => THM e | error "Expression is not a theorem"
;
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



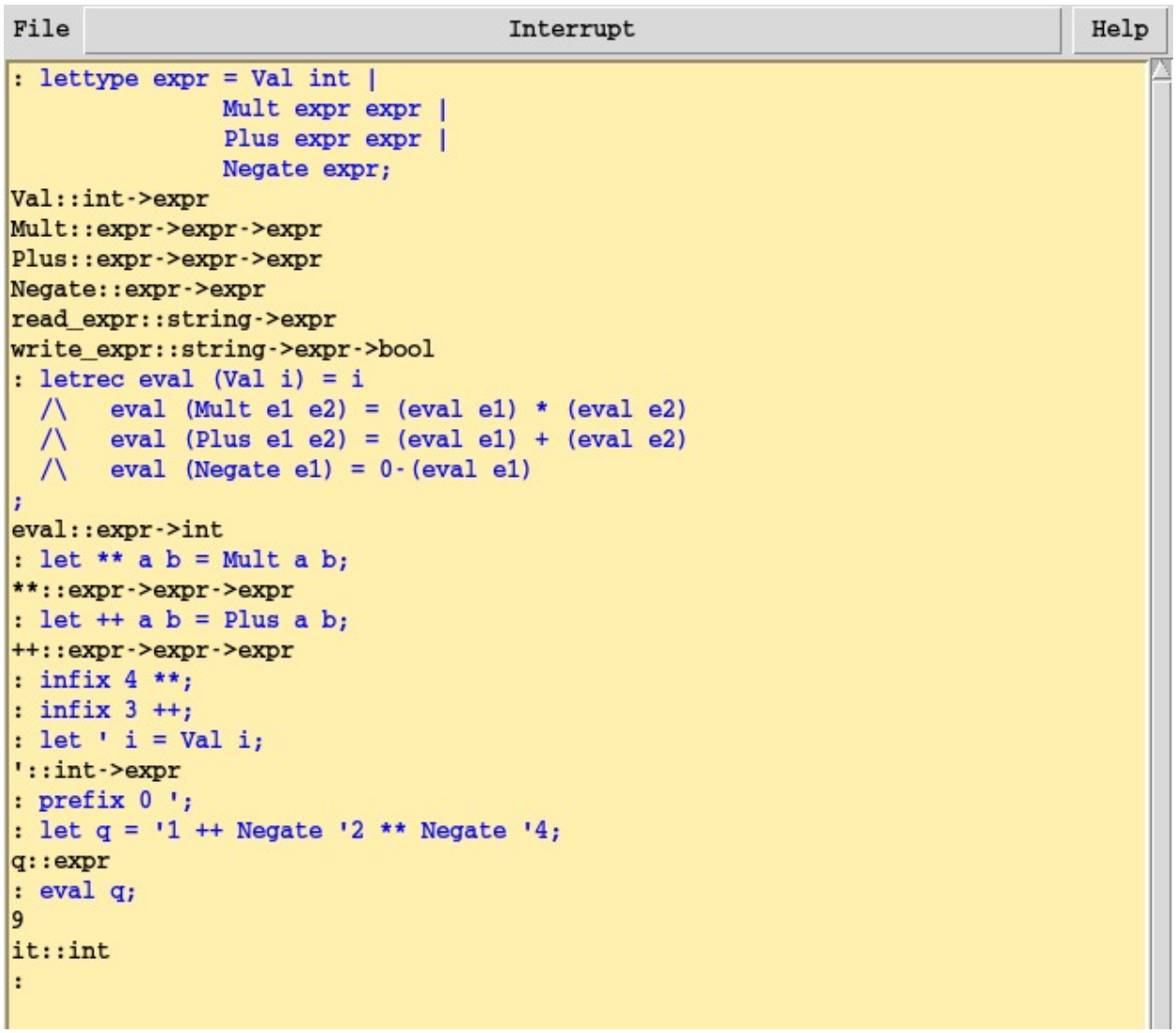
1.25 Fixity Operators

In order to make the fl code more readable, there is extensive support for changing the fixity of operators or functions.

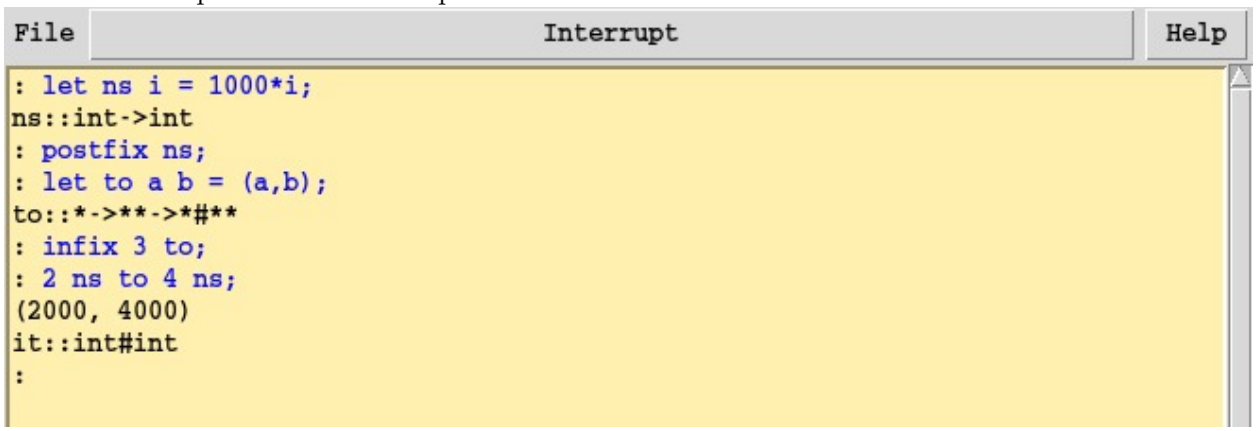
if_then_else_binder postfix nonfix prefix 0 prefix 1 binder infix 0-9 infixr 0-9 infix_unary

change the parsing of variables or

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:



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



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
:
```

1.26 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.

1.27 vossrc variables

The **VossII** system uses some user controllable settings for various behaviors. All of these can be seen by clicking on "Preference" under the "File" menu in the main fl window. In addition, on startup, **VossII** will look for a .vossrc file in the current directory or your home directory to set these preferences. For a temporary change of one of these settings, use of the GUI under Preferences is definitely recommended. However, if you want a certain setting for a particular fl program/session, there are two builtin functions that can be used: `get_vossrc` and `update_vossrc`. The first takes a name of a vossrc variable (to get the names of them, see the Preference window and click on the "copy to clipboard" icon beside the name) and returns its current value. The second changes it to a new value (and returns the old). For example:

File Interrupt Help

```
: // Read the old value
get_vossrc "DYNAMIC-ORDERING";
"YES"
it::string
: // Update the value
update_vossrc "DYNAMIC-ORDERING" "NO";
"YES"
it::string
: // Read the new value
get_vossrc "DYNAMIC-ORDERING";
"NO"
it::string
:
```

1.28 reference variables

Although fl is mostly a pure functional language, it has some non-functional parts that can be very useful. One particular non-functional part is reference variables. A reference variable is like a memory location to which one can write new values and programs can get the current value. A reference variable is created by the (psuedo-) function call ref. Ref takes a value and returns a reference to a storage location for such values and that initially contain this value. One important restriction to reference variables is that the value must be monomorphic. Thus, `ref []` is rarely usable, but, e.g., `ref {[]::(int#bool) list}` is needed. To access the current value of a reference variable, the function `deref` is used. Finally, to update the value stored in the reference variable, the infix function `:-` is used. An example of using a reference variable to count the number of times a function is called could look like:

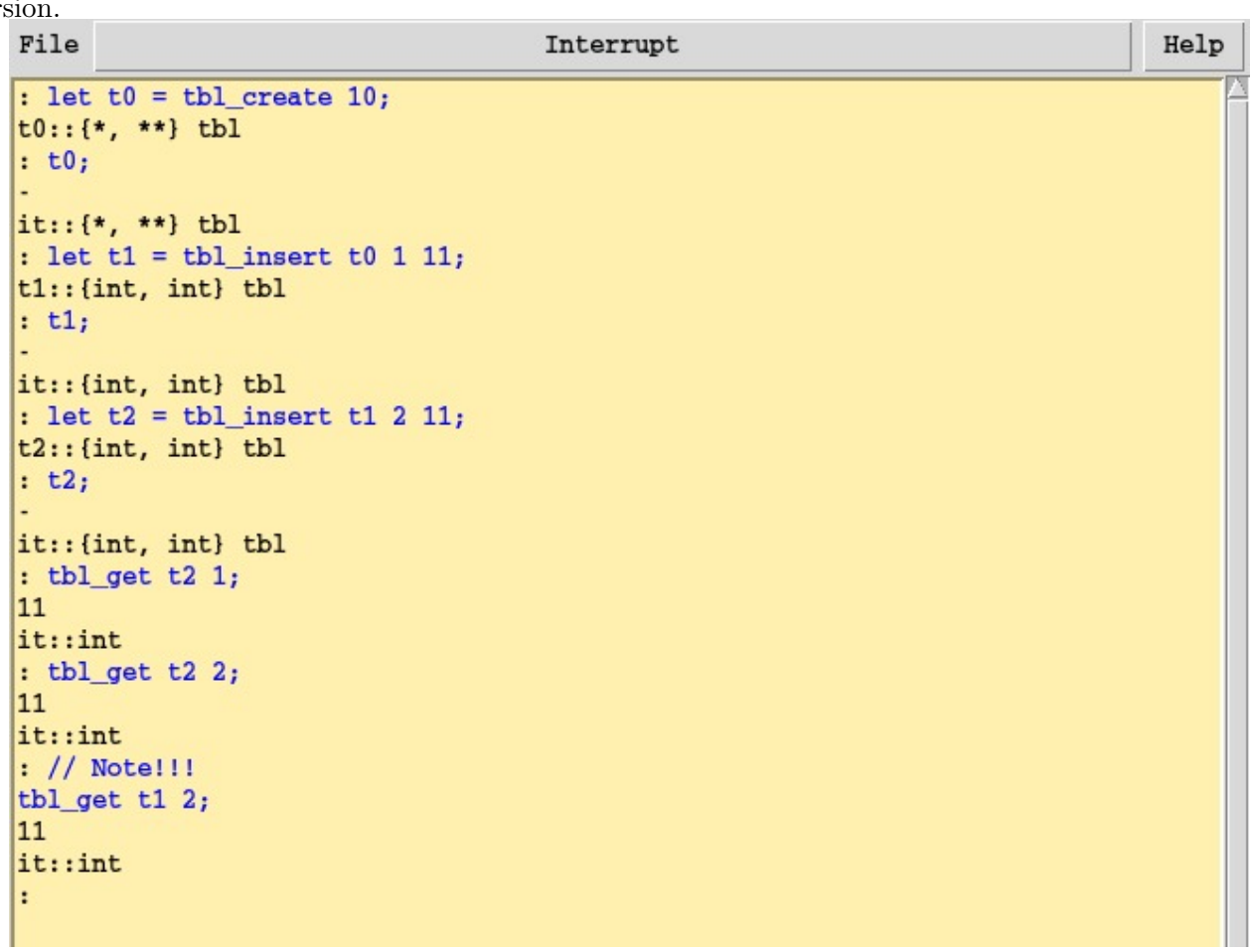
File | Interrupt | Help

```
: let cnt_r = ref 1;
cnt_r::int ref
: letrec fib 1 = 1
 /\    fib 2 = 1
 /\    fib n =
        (cnt_r := (deref cnt_r + 1)) fseq
        fib (n-1) + fib (n-2)
;
fib::int->int
: cnt_r := 0;
: fib 20;
6765
it::int
: deref cnt_r;
6764
it::int
: cletrec mfib 1 = 1
  /\    mfib 2 = 1
  /\    mfib n =
         (cnt_r := (deref cnt_r + 1)) fseq
         mfib (n-1) + mfib (n-2)
;
mfib::int->int
: cnt_r := 0;
: mfib 20;
6765
it::int
: deref cnt_r;
18
it::int
:
```

1.29 Tables

To enable fast lookup, there is a tbl type in fl. However, it has some restrictions. Although it appears to be functional, e.g., inserting a new (key,data) pair yields a new table, this is an illusion. Internally, the table is implemented as a stateful hash table and thus one always see the latest

version.




```
tbl_create     -- Make a table
tbl_insert     -- Make a "new" table by inserting a key and data
tbl_delete     -- Make a "new" table by deleting the data linked to key
tbl_member        -- Is a given key in the table
tbl_get         -- Return the data linked to key
tbl_clear     -- Empty the table
list2tbl     -- Build a table from a (key,data) list
tbl2list     -- Return the (key,data) pairs stored in the table
```

1.30 Laziness

As stated earlier, fl is a lazy language. This is mostly true, but there are some subtle exceptions. In general, laziness means two aspects: 1) an expression is never evaluated until it is needed and 2) an expression is only evaluated once. Once you introduce side effects in the language this causes problems. In particular, the second aspect may no longer be what you want. In addition, as a consequence of property 1, the order in which expressions get evaluated is difficult to predict.

To mitigate the problem of evaluation order, fl provides two mechanisms. First, the (infix) functions seq and fseq can be used to force a particular evaluation order.

Since a very common situation is that we define some expression and would like it to be evaluated before we do some more work, we could write this as:

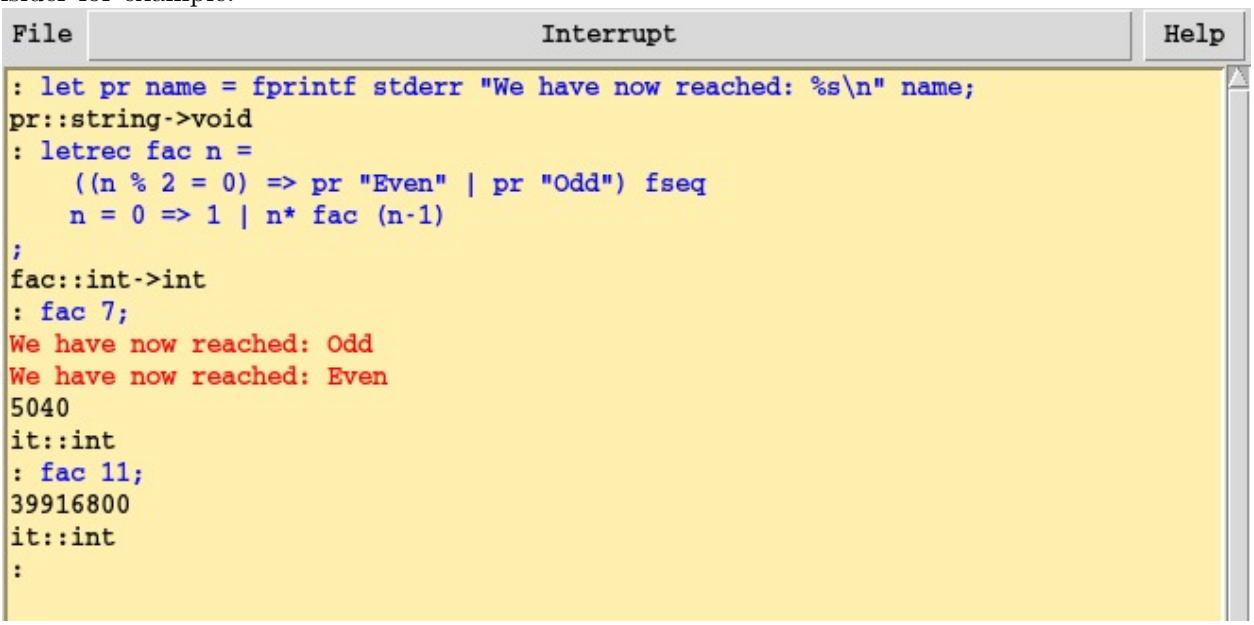
```
...
let e = <big-expression-with-side-effects> in
e fseq
...
```

However, this is so tedious that fl has introduced a short-hand for this: then and thus you can simply write:

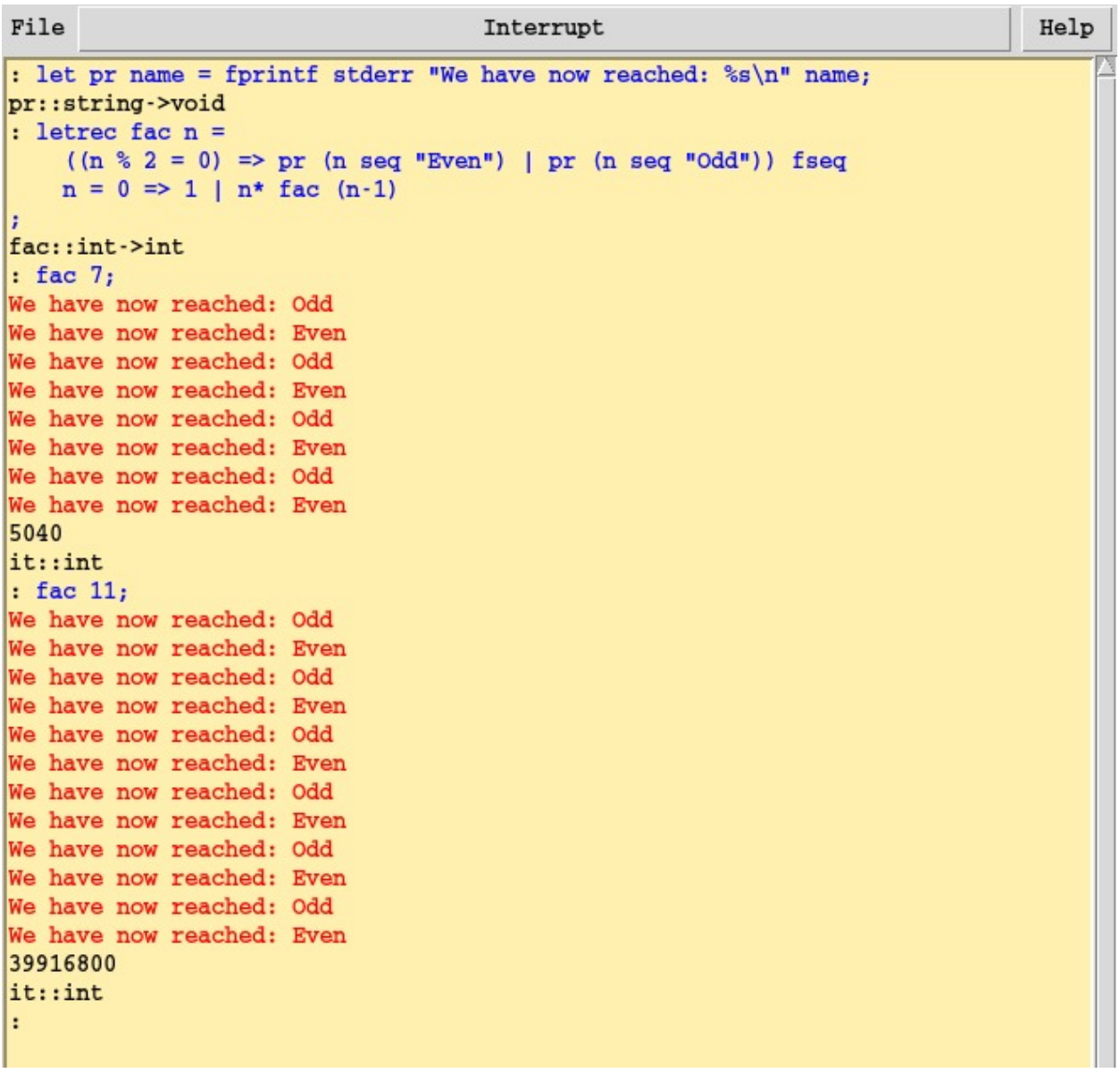
```
...
let e = <big-expression-with-side-effects> then
...
```

to achieve the same effect.

The problem with functions being evaluated once causes problems with side effecting functions. Consider for example:

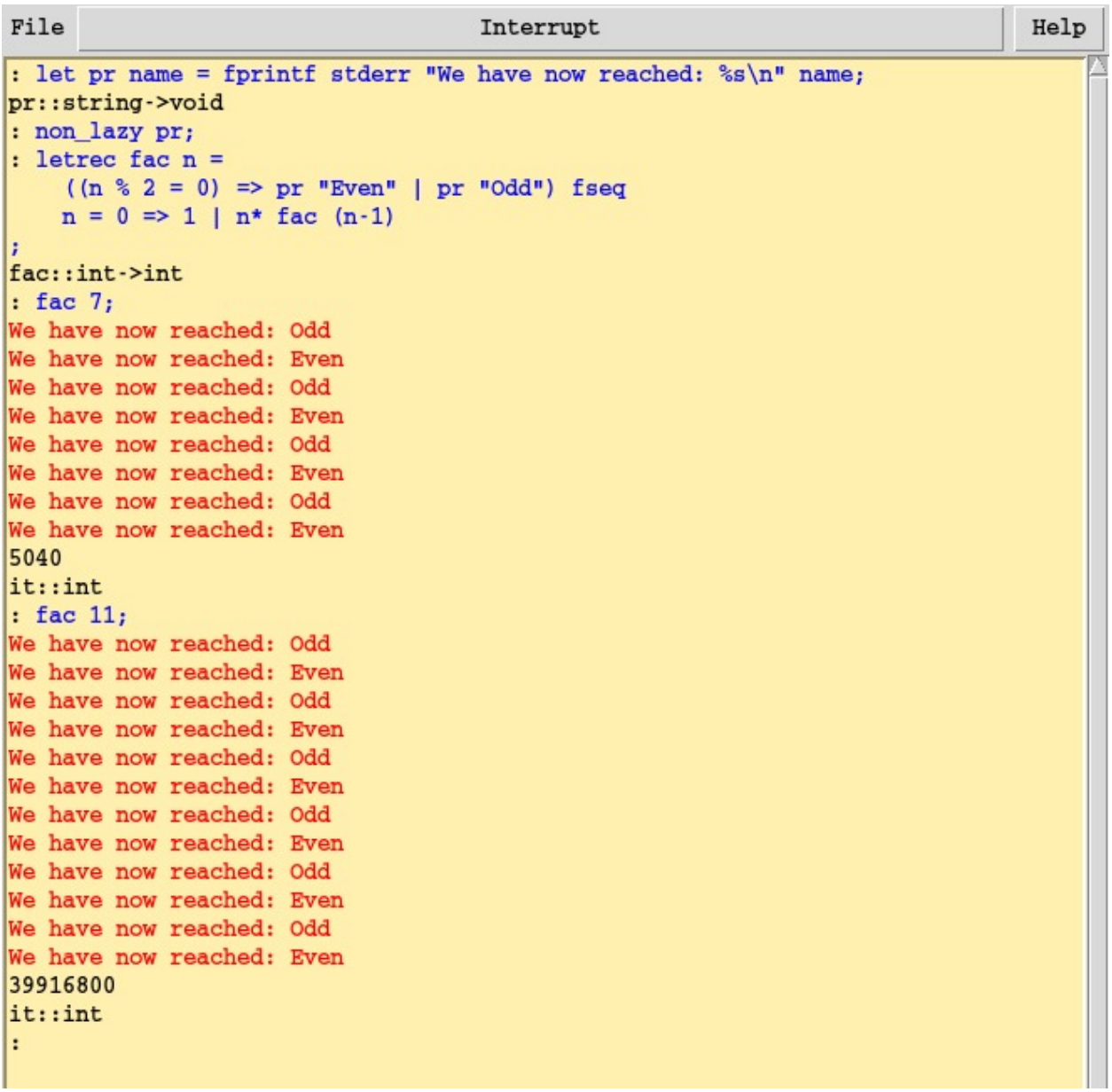


Not only do we only get two printouts for the fac 7 evaluation, we get no output the second time we call fac! The problem here is that the expressions (pr "Even") and (pr "Odd") does not depend on n and thus will only be evaluated once. What is needed is somehow introduce a dependency on n in these expressions. One approach would be:



in which we introduced a (dummy) dependency on n.

Since situation is quite common when one use side-effecting functions, fl has introduced an automatic mechanism for it. By declaring the function pr as non_lazy the compiler will insert such a dummy dependency, Thus, the above example is best written as:



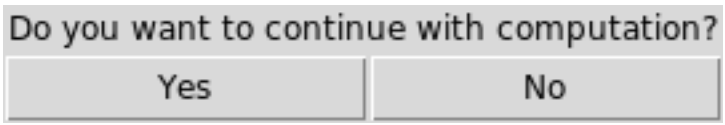
1.31 tcl interface

The fl interpreter consists of two processes. One for the main fl interpreter and one wish process for the graphical interface. As a result, there is bidirectional a link between the tcl/tk process and the fl process and thus fl can call tcl procedures and tcl procedures can call fl functions. To call tcl from fl, there is a command called tcl_eval that takes a list of strings and send them over to the running tcl interpreter. For example:

File | Interrupt | Help

```
: let go_on topic =
    let query = sprintf "Do you want to continue with %s?" topic in
    tcl_eval [
        "toplevel .foo",
        sprintf "label .foo.question -text {%s}" query,
        "frame .foo.answer",
        "button .foo.answer.yes -text Yes -command {set res 1}",
        "button .foo.answer.no  -text No  -command {set res 0}",
        "pack .foo.question -side top -fill x",
        "pack .foo.answer -side top -fill x",
        "pack .foo.answer.yes -side left -expand y -fill x",
        "pack .foo.answer.no  -side left -expand y -fill x",
        "update",
        "i_am_free",
        "tkwait variable res",
        "destroy .foo",
        "set res"
    ] = "1"
;
go_on::string->bool
:
```

If one now invokes `go_on "computation";` one would see:



and the system would wait for the user to press either. Depending on whether the user pressed Yes or No, the expression would return true or false.

Of course, in practice the tcl program is usually kept in a separate file and the tcl_eval command simply contains a tcl command to source that file.

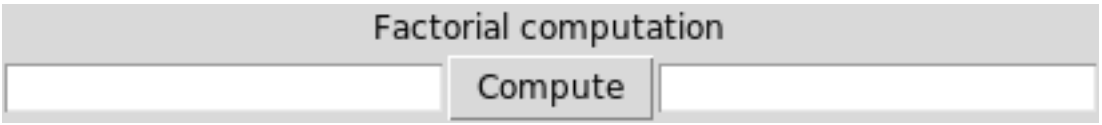
One thing to note is that the main **VossII** window is in a looked (busy-wait) state during evaluations of a tcl_eval. To "unlock" it, the tcl command "i_am_free" is needed before waiting for the variable to change.

To have tcl call fl is even easier. If you declare an fl function of suitable type (more on that later) and add the directive `export_to_tcl` directly after, a tcl procedure will exist with that name and that will call the fl function when invoked. For example:

File | Interrupt | Help

```
: letrec fl_fac n = (n = 1) => 1 | n*fl_fac (n-1);
fl_fac::int->int
: export_to_tcl fl_fac;
: let fac_calc title =
    tcl_eval [
        "toplevel .foo",
        sprintf "label .foo.title -text {%s}" title,
        "frame .foo.c",
        "entry .foo.c.inp",
        "button .foo.c.do -text Compute -command { docomp }",
        "entry .foo.c.out",
        "pack .foo.title .foo.c -side top -fill x",
        "pack .foo.c.inp .foo.c.do .foo.c.out -side left",
        (
            "proc docomp {} {"^
            "   set i [.foo.c.inp get];"^
            "   set o [fl_fac $i];"^
            "   .foo.c.out delete 0 end;"^
            "   .foo.c.out insert 0 $o;"^
            "}"
        )
    ]
;
fac_calc::string->string
:
```

If one now evaluates `fac_calc "Factorial computation";`, one would see:

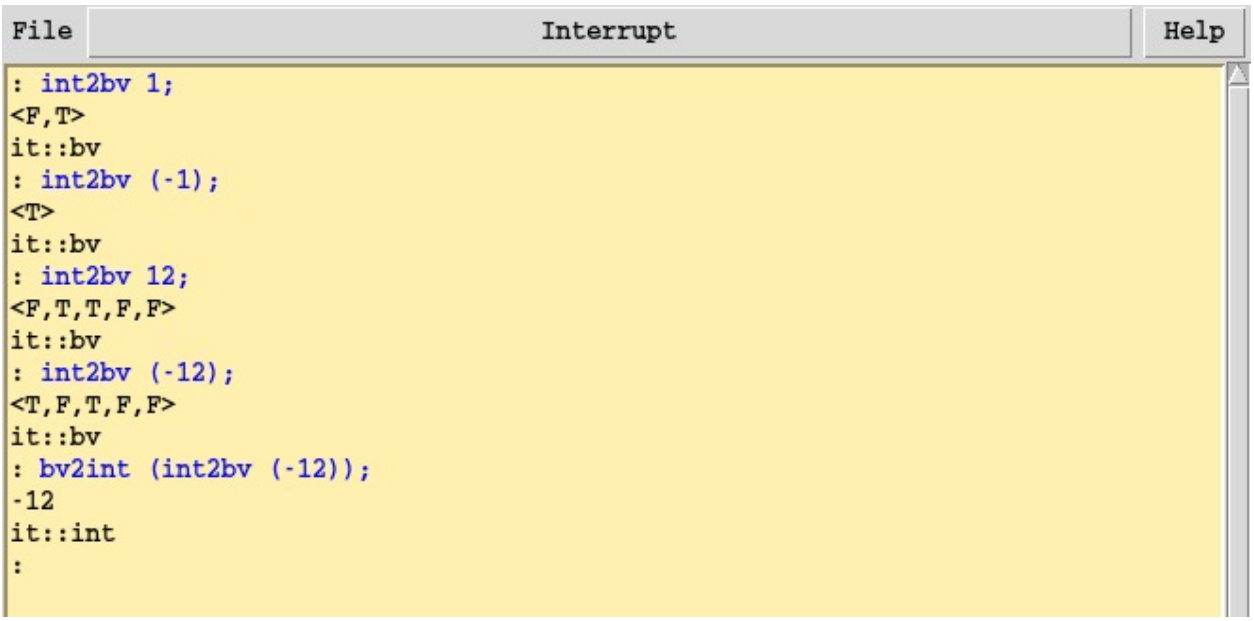


Note that the tcl procedure (proc docomp) has to be given as a single string in tcl_eval. Again, putting this code in a separate tcl file is far easier.

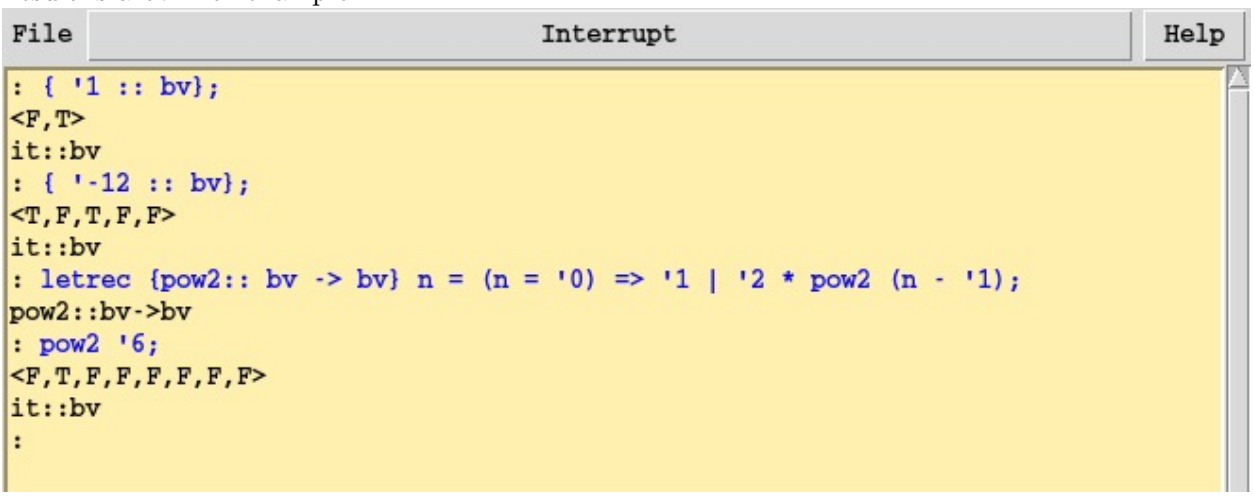
Finally, although not a requirement, the convention is to name fl functions that will be used as callback functions in tcl with a prefix `fl_`.

1.32 Bitvectors

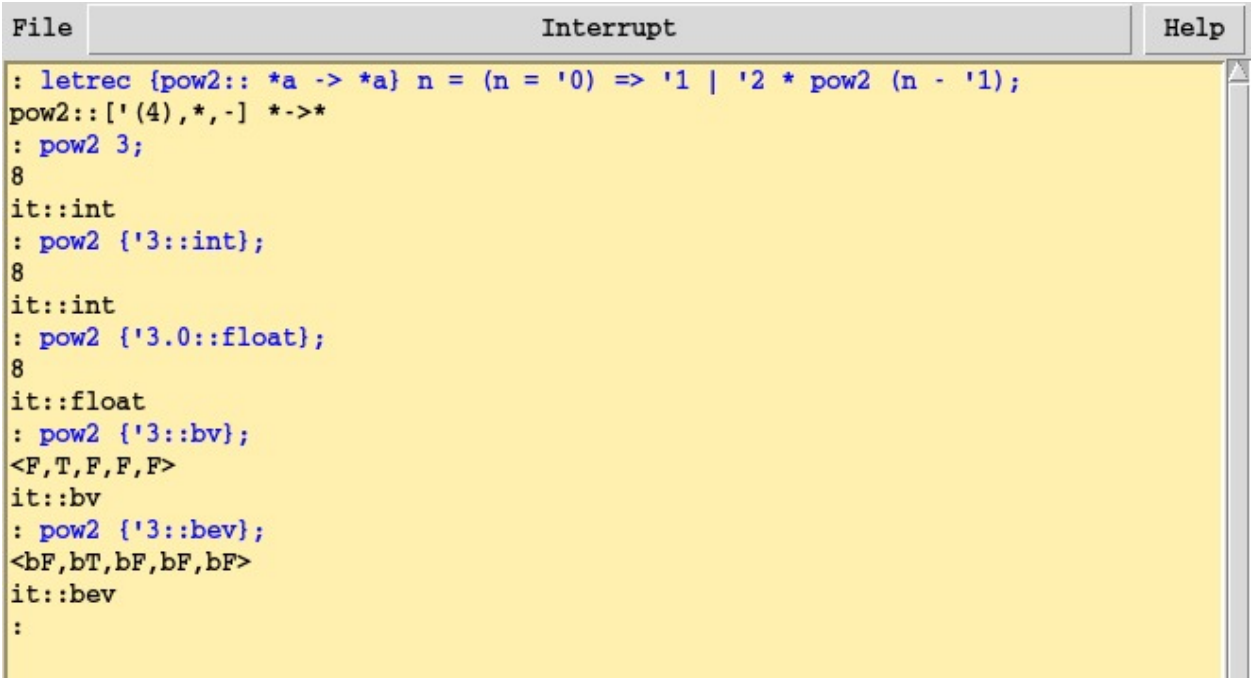
To ease writing specifications and creating code for bitvector arithmetic, a builtin type "bv" has been introduced. A bv is an arbitrary precision two's complement bitvector that grows and shrinks as needed. It is effectively a list of BDDs. One should interpret a bv as an infinite bitvector where the most significant digit (head of the list) is repeated infinitely many times. To illustrate:



To make it easier to create bitvectors for constants, an overloaded function ' is available. Of course, since it is an overloaded function, a context (or type annotations) must be used to ensure the result is a bv. For example:

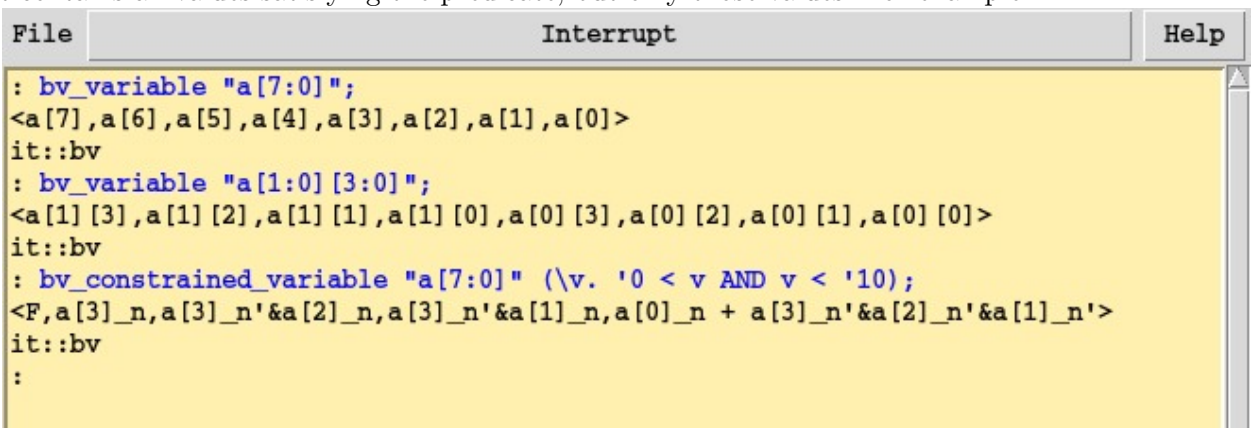


Of course, the pow2 function could be defined for all the types that the basic operators are defined over. Thus:



shows how to define pow2 so that it can be used for ints, floats, bvs, and bevs (like bvs but using bexprs instead of BDDs for the elements in the list). Note that we need to provide a type annotation `(*a -> *a)` in the declaration since otherwise the system would not be able to resolve the overloading without both the input type and the output type.

To create a vector with variables, there are two functions: bv_variable and bv_constrained_variable. Both of these functions take a string as the name of the vector. This string defines the size of the vector and multi-dimensional vectors can also be created. The bv_constrained_variable function takes a second argument whith should be a predicate that given a bitvector returns a true/false for the values that are of interest. Here it will return a bitvector with a parametric representation that contains all values satisfying the predicate, but only those values. For example:



An important point to remember is that bitvectors are signed. Thus, if an unsigned version is needed, one has to use the constrained version. Alternatively, one can also use the bv_ZX, to zero extend (and thus make positive) the bitvector. Of course, one has to be careful to get the right size of the resulting bv. For example, if an unsigned bitvector with 8 variables is needed, the following two definitions are equivalent (although they will use slightly different variable names).

```
File    Interrupt    Help
: bv_constrained_variable "a[8:0]" (\v. v >= '0);
<F,a[7]_n,a[6]_n,a[5]_n,a[4]_n,a[3]_n,a[2]_n,a[1]_n,a[0]_n>
it::bv
: bv_ZX (bv_variable "a[7:0]");
<F,a[7],a[6],a[5],a[4],a[3],a[2],a[1],a[0]>
it::bv
:
```

Sometimes, it is useful to get access to the list of the bitvector. For that purpose, there is a bv2list and a list2bv function. If you want a specific length of the list, fixed_bv2list takes a length argument and a sign/zero extension flag in addition to the bitvector and returns a sign- or zero-extended list of the desired length. Of course, if the length is too small for the bitvector to fit, the function will raise an exception.

```
File    Interrupt    Help
: bv2list (int2bv (-12));
[T,F,T,F,F]
it::bool list
: list2bv [F,F,F,T,T];
<F,T,T>
it::bv
: fixed_bv2list 16 F '-3;
[F,F,F,F,F,F,F,F,F,F,F,F,T,F,T]
it::bool list
: fixed_bv2list 16 T '-3;
[T,T,T,T,T,T,T,T,T,T,T,T,T,F,T]
it::bool list
:
```

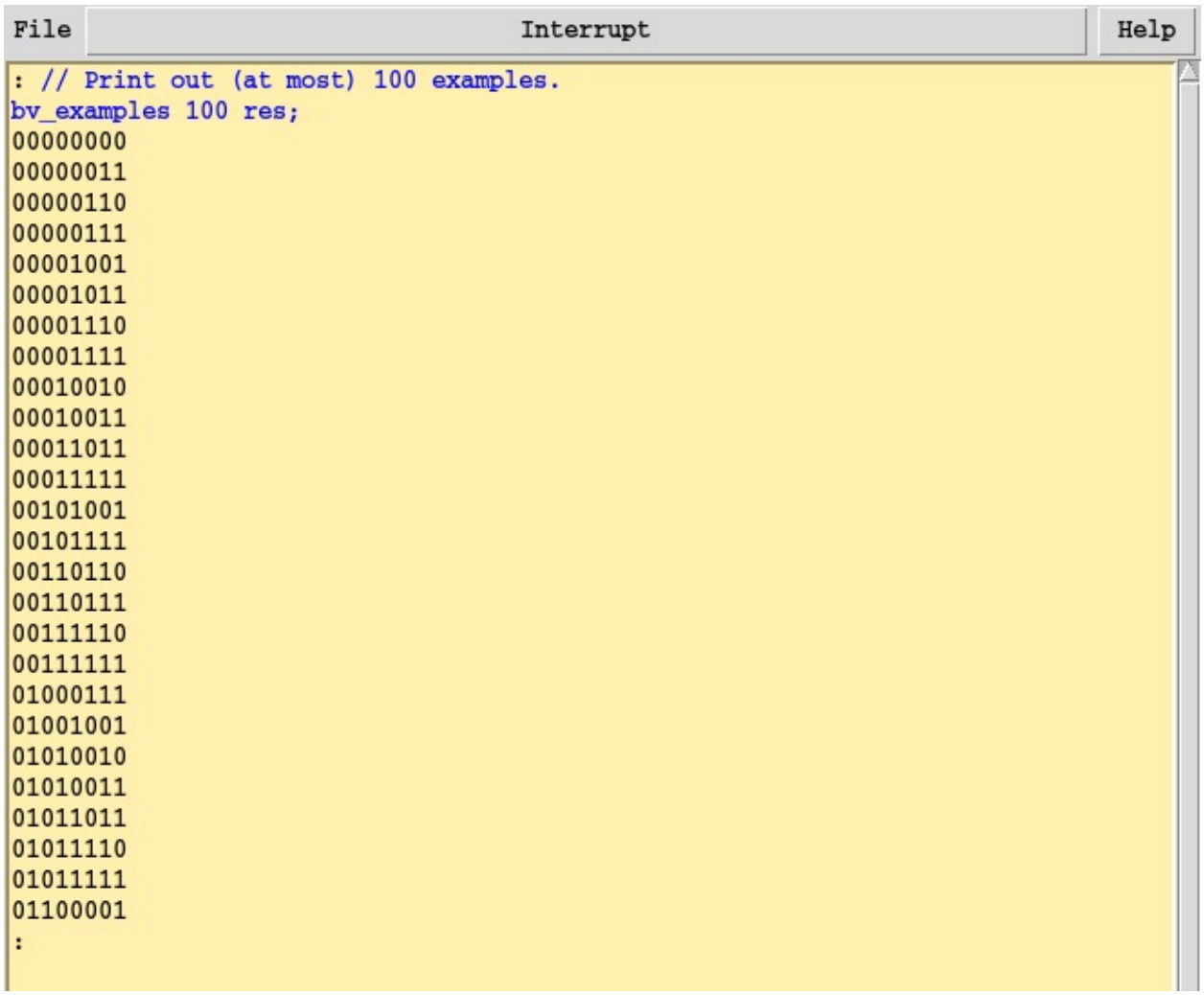
Finally, in order to understand a (symbolic) bitvector, the bv_example function can be very useful. For example:

```
File                                    Interrupt                                    Help
: // Return an integer n if its collatz sequence is longer than n.
// Otherwise, return 0
let {large_collatz:: bv -> bv} n =
    letrec collatz cnt cur = cur = '1 => cnt |
            (cur % '2) = '1 => collatz (cnt + '1) ('3 * cur + '1)
                             | collatz (cnt + '1) (cur / '2)
    in
    let res = collatz '0 n in
    n < res => n | '0
;
large_collatz::bv->bv
: // Try this for integers between 1 and 99.
let res =
    large_collatz (bv_constrained_variable "a[7:0]" (\v. '1 <= v AND v < '100))
;
res::bv
:
```

If we simply evaluated res, we would get:

```
File                                    Interrupt                                    Help
: res;
<F,a[6]_n&a[5]_n&a[1]_n'&a[0]_n + a[6]_n&a[5]_n'&a[4]_n&a[3]_n&a[2]_n&a[1]_n + a
[6]_n&a[5]_n'&a[4]_n&a[2]_n'&a[1]_n&a[0]_n + a[6]_n&a[5]_n'&a[4]_n&a[3]_n'&a[2]_
n'&a[1]_n + a[6]_n&a[4]_n'&a[3]_n&a[2]_n'&a[1]_n'&a[0]_n OR ... ,a[6]_n&a[5]_n&a
[1]_n'&a[0]_n + a[6]_n'&a[5]_n&a[4]_n&a[2]_n&a[1]_n + a[6]_n'&a[5]_n&a[3]_n&a[2]
_n&a[1]_n&a[0]_n + a[5]_n&a[4]_n'&a[3]_n&a[2]_n'&a[1]_n'&a[0]_n,a[6]_n&a[5]_n'&a
[4]_n&a[3]_n&a[2]_n&a[1]_n + a[5]_n'&a[4]_n&a[2]_n'&a[1]_n&a[0]_n + a[5]_n'&a[4]
_n&a[3]_n'&a[2]_n'&a[1]_n + a[6]_n'&a[5]_n&a[4]_n&a[2]_n&a[1]_n + a[5]_n'&a[4]_n
&a[3]_n&a[1]_n&a[0]_n OR ... ,a[6]_n&a[5]_n'&a[4]_n&a[3]_n&a[2]_n&a[1]_n + a[5]_
n'&a[4]_n&a[3]_n&a[1]_n&a[0]_n + a[5]_n'&a[4]_n'&a[3]_n&a[2]_n'&a[1]_n'&a[0]_n +
 a[6]_n'&a[5]_n&a[4]_n&a[3]_n&a[2]_n&a[1]_n + a[6]_n'&a[3]_n&a[2]_n&a[1]_n&a[0]_
n OR ... ,a[6]_n&a[5]_n'&a[4]_n&a[3]_n&a[2]_n&a[1]_n + a[5]_n'&a[4]_n'&a[3]_n'&a
[2]_n&a[1]_n&a[0]_n + a[6]_n'&a[5]_n&a[4]_n&a[2]_n&a[1]_n + a[6]_n'&a[3]_n&a[2]_
n&a[1]_n&a[0]_n OR ... ,a[6]_n&a[5]_n'&a[4]_n&a[3]_n&a[2]_n&a[1]_n + a[5]_n'&a[4
]_n&a[2]_n'&a[1]_n&a[0]_n + a[5]_n'&a[4]_n&a[3]_n'&a[2]_n'&a[1]_n + a[5]_n'&a[4]
_n'&a[3]_n'&a[2]_n&a[1]_n&a[0]_n + a[6]_n'&a[5]_n&a[4]_n&a[2]_n&a[1]_n OR ... ,a
[6]_n&a[5]_n&a[1]_n'&a[0]_n + a[5]_n'&a[4]_n&a[3]_n&a[1]_n&a[0]_n + a[5]_n'&a[4]
_n&a[2]_n'&a[1]_n&a[0]_n + a[4]_n'&a[3]_n&a[2]_n'&a[1]_n'&a[0]_n + a[5]_n'&a[4]_
n'&a[3]_n'&a[2]_n&a[1]_n&a[0]_n OR ... >
it::bv
:
```

However, this is (almost) impossible to understand. To help understand these symbolic bitvectors, it is often useful to ask for explicit examples. Thus, we could do:



which is much easier to understand.

As already aluded to, all the usual arithmetic, logic, and comparison operations are aviable for bitvectors. Many, if not most, are also overloaded to the reguler operations. To see all available bv functions, use the help system and restrict the argument type to match bv, or the result type to match bv.

2 Hardware Models

An important part of the **VossII** system is its symbolic trajectory evaluation (STE) formal verification engine. Intuitively, STE is a generalization of a symbolic simulator in which the Boolean domain has been extended to a lattice structure and thus combines symbolic varaibles/expressions and lattice operations. For the STE system to work, a model of some hardware is needed. Currently, **VossII** provides two ways of getting such a model: 1) by reading in a Verilog description using, a slightly extended, Yosys [**?**] system or define the model directly in an extension of fl, called HFL. Either way, an fl datatype called pexlif will be created first. This datatype captures hiearchy, wire names, and RTL level assignments (both instantaneous as well as delayed) of the hardware. For further use, the pexlif is often compiled to a more efficient model, called fsm. We will return to this later.

2.1 Pexlif

The pexlif datatype is meant to be used to represent hardware modelled at a register transfer level. In other words, a pexlif model captures the design hierarchy (including the wire names) as well as the zero-delay combinational parts and the phase delay parts. More specifically, the pexlif (and associated types) are defined as:

```
lettype pexlif =
        PINST
            {name::string}
            {attrs::(string#string) list}
            {leaf::bool}
            {fa_inps:: (string#(string list)) list}
            {fa_outs:: (string#(string list)) list}
            {internals:: string list}
            {content::content}
andlettype content =
            P_HIER {children::pexlif list}
        |   P_LEAF {fns::update_fn list}
;
```

and

```
lettype update_fn =
            W_UPDATE_FN {lhs::wexpr} {rhs::wexpr}
        |   W_PHASE_DELAY {lhs::wexpr} {rhs::wexpr}
;
```

The word-level expressions (wexpr) are defined as:

```
lettype wexpr =
            W_X {sz::int}
        |   W_CONST {sz::int} {v::int}
        |   W_NAMED_CONST {name::string} {sz::int} {v::int}
        |   W_VAR {sz::int} {base::string}
        |   W_EXPLICIT_VAR {sz::int} {name::string}
        |   W_AND {a::wexpr} {b::wexpr}
        |   W_OR {a::wexpr} {b::wexpr}
        |   W_NOT {a::wexpr}
        |   W_EQ {a::wexpr} {b::wexpr}
        |   W_PRED {name::string} {cond::wexpr}
        |   W_GR {a::wexpr} {b::wexpr}
        |   W_ADD {a::wexpr} {b::wexpr}
        |   W_SUB {a::wexpr} {b::wexpr}
        |   W_MUL {a::wexpr} {b::wexpr}
        |   W_DIV {a::wexpr} {b::wexpr}
        |   W_MOD {a::wexpr} {b::wexpr}
        |   W_SHL {a::wexpr} {b::wexpr}
        |   W_SHR {a::wexpr} {b::wexpr}
        |   W_ASHR {a::wexpr} {b::wexpr}
        |   W_SX  {sz::int}  {w::wexpr}
        |   W_ZX  {sz::int}  {w::wexpr}
        |   W_ITE {cond::wexpr} {t::wexpr} {e::wexpr}
        |   W_SLICE {indices::int list} {w::wexpr}
        |   W_NAMED_SLICE {name::string} {indices::int list} {w::wexpr}
        |   W_CAT {parts::wexpr list}
        |   W_MEM_READ {info::mem} {mem::wexpr} {addr::wexpr}
        |   W_MEM_WRITE {info::mem} {mem::wexpr} {addr::wexpr} {data::wexpr}
;
```

where

```
lettype mem = MEM
                {addr_size::int}
                {lines::int}
                {data_size::int}
;
```

and is used to represent the various expressions in an RTL model.

To illustrate a simple function operating over a pexlif, here is one computing the number of (internal) wires in a model:

File Interrupt Help

```
: let get_children (P_HIER ch) = ch
 /\ get_children (P_LEAF _) = []
;
get_children::content->(pexlif list)
: letrec number_of_wires (PINST name opts leaf fa_inps fa_outs ints content) =
    let local = length (md_expand_vectors ints) in
    leaf => local |
    itlist (\pp. \r. number_of_wires pp+r) (get_children content) local
;
number_of_wires::pexlif->int
:
```

A more interesting function might be one that extracts a sub circuit based on its name:

File Interrupt Help

```
: letrec extract_subckts target p =
    val (PINST name opts leaf fa_inps fa_outs ints content) = p in
    str_is_suffix target name => [p] |
    leaf => [] |
    flatmap (extract_subckts target) (get_children content)
;
extract_subckts::string->pexlif->(pexlif list)
:
```

Here we used only the sufix of the name to match.

Our final example is a typical example of functions used to manipulate the circuit hierarchy. It takes a predicate and a pexlif and flattens the pexlif as long as the stop-predicate does not evaluate to true.

File | Interrupt | Help

```
: let flatten_pexlif stop_pred p =
    letrec do_flat sub pref p =
        let tr n = (assoc n sub) catch n in
        val (PINST name attrs leaf fa_inps fa_outs ints content) = p in
        leaf OR stop_pred p =>
            let tr_io (f,as) =
                let as' = md_merge_vectors (map tr (md_expand_vectors as)) in
                (f,as')
            in
            let fa_inps' = map tr_io fa_inps in
            let fa_outs' = map tr_io fa_outs in
            [(PINST name attrs leaf fa_inps' fa_outs' ints content)]
        |
            val (P_HIER children) = content in
            let mk_io_sub (f,as) =
                zip (md_expand_vector f) (map tr (md_expand_vectors as))
            in
            let io_sub = flatmap mk_io_sub (fa_inps @ fa_outs) in
            let mk_int_sub f =
                map (\n. n, sprintf "%s%s" pref n) (md_expand_vector f)
            in
            let int_sub = flatmap mk_int_sub ints in
            let sub' = io_sub @ int_sub in
            let prefs = map (sprintf "%si%d/" pref) (1 upto length children) in
            flat (map2 (do_flat sub') prefs children)
    in
    let children = do_flat [] "" p in
    let wires_used (PINST _ _ _ fa_inps fa_outs _ _) =
        md_expand_vectors (flatmap snd (fa_inps @ fa_outs))
    in
    let all = setify (flatmap wires_used children) in
    val (PINST name attrs _ fa_inps fa_outs _ _) = p in
    let ios = md_expand_vectors (flatmap snd (fa_inps @ fa_outs)) in
    let new_ints = md_extract_vectors (all subtract ios) in
    (PINST name attrs F fa_inps fa_outs new_ints (P_HIER children))
;
-Loading file /tmp/voss2_cseger_o4I89F/inp_XXXXXX_oxsJj3:
T
it::bool
:
```

2.2 HFL

In order to model hardware efficiently and with type safety, a deeply embedded hardware description language, called HFL, has been created. Since it is deeply embedded in fl, one can use many/most of fl's mechanisms for describing the hardware. For example, it is quite easy to define a hardware structure recursively.

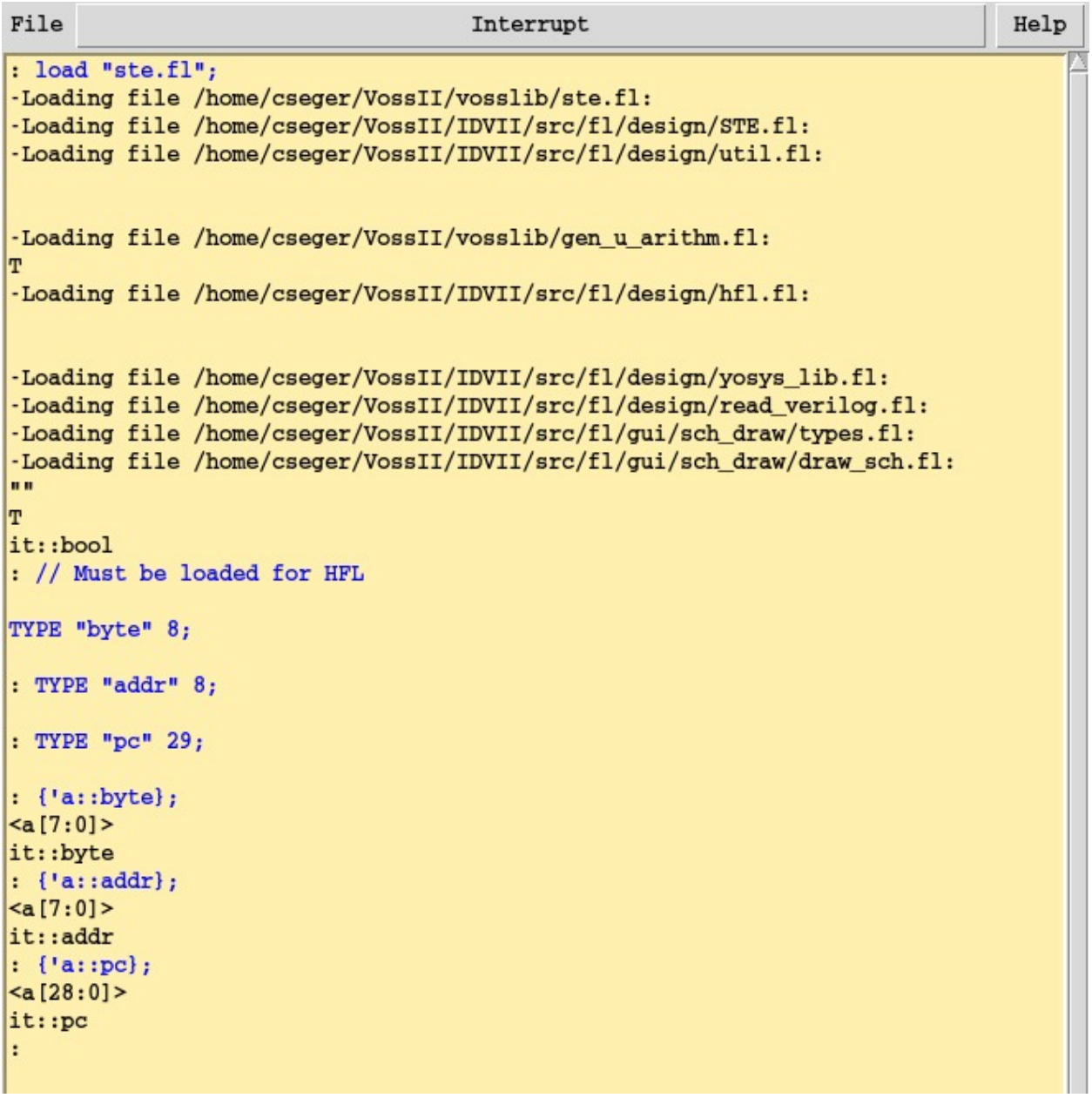
There are two parts to HFL: 1) a set of functions to create hardware types, and 2) functions (many with special binder fixity) for describing the hardware.

2.2.1 HFL Types

A hardware type is effectively a algebraic datatype in fl for which a number of destructor, constructor, and misc. other functions have been created. There are four basic ways of creating a hardware type: TYPE, ENUM, STRUCT, and MEMORY. The ENUM and STRUCT have two versions depending on how much control is needed to define the encoding. More precisely:

```
TYPE typename size

ENUM typename [name]
ENUM typename [(name,encoding)]

STRUCT typename [(fieldname,type)]
STRUCT typename [(fieldname,type,indices)]

MEMORY typename [size] wordtype
```

To illustrate how to declare such types and how to extract/update fileds, consider the following simple example:



Here we declared three simple hardware types for vectors of size 4 and 8 respectively. Note that although both byte and addr are of size 8, they are different types and thus

```
File                    Interrupt                    Help
: {'1::byte} '+' {'1::byte};
0x2
it::byte
: // Ok
{'1::byte} '+' {'1::addr};
===Type mismatch: addr and byte
===Type error around line 15
Inferred type is:
        byte
but its usage requires it to be of type:
        addr
: // Fails typechecking
```

To continue the example with the two types of ENUM:

```
File                    Interrupt                    Help
: ENUM "op" ["ADD", "JMP", "NOP"];

: ENUM "state" [("INIT", 1), ("FETCH", 2), ("EXECUTE", 4), ("WB", 8)];

: {'a::op};
<a[1:0]>
it::op
: {'a::state};
<a[3:0]>
it::state
: ADD;
0x0
it::op
: NOP;
0x2
it::op
: INIT;
0x1
it::state
: WB;
0x8
it::state
: is_FETCH 'a;
<a[3]'&a[2]'&a[1]&a[0]'>
it::bit
: is_WB 'a;
<a[3]&a[2]'&a[1]'&a[0]'>
it::bit
:
```

where the second type gives an explicit encoding. Note that it is valid to give the same encoding to two different constants (although definitely not recommended!). Note also that evaluating this command will create (zero-arity) functions for the enumerated constants. In addition, there will

be one predicate, named is_¡constant¿ for every constant declared. Note that these functions are overloaded (if needed) so that two differnt ENUMs can have the same named constants. It is up to the typechecking to resolve the usage so that the correct encoding is used. Sometimes, this will require a typecast.

Since enumerated types do not have any numerical value, it is an error to try to perform aritmetic operations on them. For example:

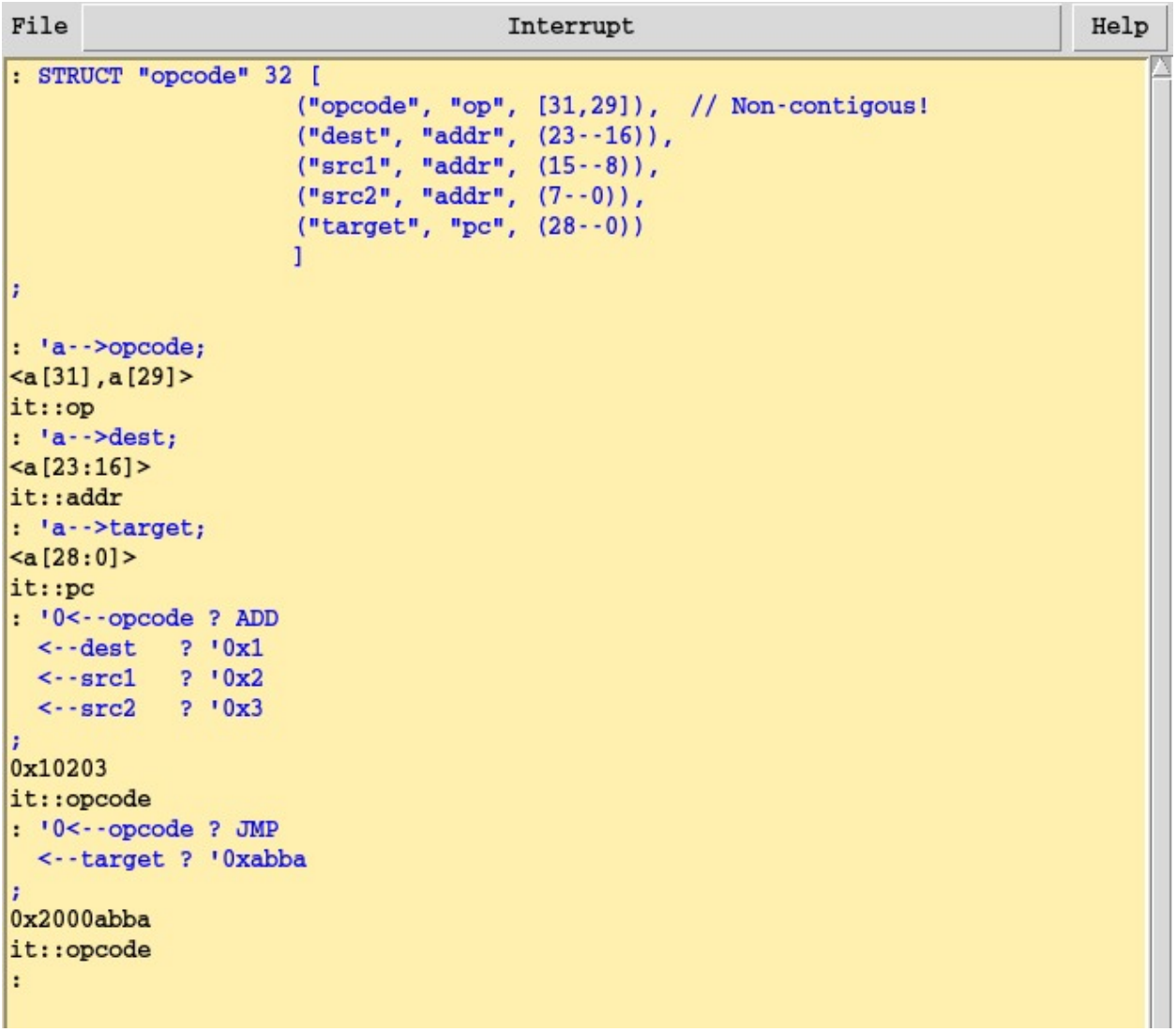
```
File                    Interrupt                    Help
: ADD '+' JMP;
Failure:    Cannot perform + on vectors of type op

Stack trace:
1:      wADD

it::op
:
```

fails. (Note that since fl does not have dependent types, the failure is a runtime failure triggered when the pexlif is built.)

The STRUCT function creates a bitvector as well as accessor functions for the different fields. The first version of STRUCT is simply useful when creating "bundles" of other vectors. For example:

```
File                    Interrupt                    Help
: STRUCT "bundle" [("high","byte"),("low","byte")];

: 'a-->high;
<a[15:8]>
it::byte
: // Extract field
'a<--high ? '0xab;
<1,0,1,0,1,0,1,1,a[7:0]>
it::bundle
: // Update field
'a<--high ? '0xab <-- low ? '0xba;
0xabba
it::bundle
: // Nested updates
```

The more general STRUCT function allows the fields to overlap. Since the system cannot easily determine the size of the resulting type, the user has to give the size. Also, each field now consists of three items: the name of the field, the type of the field, and a list of indices in the vector that this field refers to. For example:



Finally, the MEMORY construct is used to create, possibly multi-dimensional, arrays. When a MEMORY type "foo" is created, there are four additional functions created:

```
read_foo  mem:: foo addr_1 addr_2 ... addr_n
write_foo mem:: foo addr_1 addr_2 ... addr_n new_data
Read_foo  mem:: foo addr_1 addr_2 ... addr_n out_name
Write_foo mem:: foo addr_1 addr_2 ... addr_n new_data new_mem
```

The read_foo takes a memory and a list of addresses (one for each dimension of the memory) and returns the value in that memory location. The write_foo takes a memory and a list of addresses and a new data value and returns the new memory in which the data at the selected address has been replaced with the new_data. The Read_foo and Write_foo functions are functions that create a pexlif for the corresponding operation (and thus has a better visualization). We will return to these later.

A simple example of MEMORY could be:

```
File                    Interrupt                    Help
: MEMORY "regfile" [(10,"addr")] "byte";

T
it::bool
: {'a::regfile};
<a[9:0][7:0]>
it::regfile
: read_regfile 'a '0x3;
<a[3][7:0]>
it::byte
: write_regfile 'a '0x0 '0xab;
<a[9:1][7:0],1,0,1,0,1,0,1,1>
it::regfile
: write_regfile 'a '0x3 'd;
<a[9:4][7:0],d[7:0],a[2:0][7:0]>
it::regfile
:
```

A more complex example:

```
File                    Interrupt                    Help
: MEMORY "array" [(4,"addr"),(5,"addr")] "byte";

T
it::bool
: {'a::array};
<a[3:0][4:0][7:0]>
it::array
: read_array 'a '0x3 '0x1;
<a[3][1][7:0]>
it::byte
: write_array 'a '0x0 '0x0 '0xab;
<a[3:1][4:0][7:0],a[0][4:1][7:0],1,0,1,0,1,0,1,1>
it::array
: write_array 'a '0x3 '0x2 'd;
<a[3][4:3][7:0],d[7:0],a[3][1:0][7:0],a[2:0][4:0][7:0]>
it::array
:
```

2.2.2 HFL Expressions

To make it more convenient to describe combinational circuits a small embedded DSL has been created for expressions as well. We have already shown some examples. Basically, the folloing operators are availble

```
Constants:
'23                 // Decimal number
'0x2fc00            // Hexadecimal number
'0b10010            // Binary number
'X                  // Unknown (Don't care) value

Variables:
'a                  // A singel bit or simple vector depending on return type

Logical expressions:
'~ e              // (Bitwise) negation
e1 '&' e2           // (Bitwise) AND
e1 '|' e2           // (Bitwise) OR
e1 '^' e2           // (Bitwise) XOR
e1 '<<' e2          // Shift-left
e1 '>>' e2          // Logical shift right
e1 '|>>' e2         // Arithmetic shift right
c '?' t ':' e       // If-then-else expression

Arithmetic expressions:
ZX e                // Zero extension
SX e                // Sign extension
e1 '+' e2           // Addition
e1 '-' e2           // Subtraction
e1 '*' e2           // Multiplication
e1 '/' e2           // Division
e1 '%' e2           // Modulus

Misc. operations:
e1 ++ e2            // Catenate two bitvectors
e1    i             // Select a single bit
e1     l            // Select a range of bits
IF c THEN t ELSE e  // Alternative if-then-else
```

2.2.3 HFL Hierarchy Creation

In order to manage the complexity of a design, almost all hardware is defined in a hierarchical manner. Thus, HFL provides a convenient way of defining modules and instantiating such modules.

A module is generally defined as:

```
let module_name =
    // Inputs
    <type>_input    i1.
    ...
    <type>_input    ik.
    // Outputs
    <type>_output   o1.
    ...
    <type>_output   ol.
    // Internal signals
    <type>_internal m1.
    ...
    <type>_internal mn.
    CELL "xyz" [
<assignments>
<instantiations>
];
```

Note that when a hardware type "foo" is created, there are (at least) three additional functions

created: foo_input, foo_output, and foo_internal, that are binder functions that makes it wasy to declare inputs/outputs or internal signals of this type. However, there are also an input, output, and internal functions available in which the signals declares will not be forced to have a particular type. Depending on the context and use, the type checker will then possibly restrict the use further. Finally, there is also a list_input and list_output functions available to declare signals that are (unrestricted) lists of signals. In addition, there is a list_internal function that takes an initial integer expression that denotes how many signals should be in this list of internal signals.

For example:

File Interrupt Help

```
: let add3 =
    byte_input     a.
    byte_input     b.
    byte_input     c.
    byte_output     o.
    byte_internal   m.
    CELL "draw_hier add3" [
        m <- a '+' b,
        o <- m '+' c
];
add3::byte->byte->byte->byte->pexlif
:
```

which might be used as:

File Interrupt Help

```
: let add9 =
    byte_input     i1 i2 i3 i4 i5 i6 i7 i8 i9.
    byte_output     o.
    byte_internal   m1 m2 m3.
    CELL "draw_hier add6" [
        add3 i1 i2 i3 m1,
        add3 i4 i5 i6 m2,
        add3 i7 i8 i9 m3,
        add3 m1 m2 m3 o
];
add9::byte->byte->byte->byte->byte->byte->byte->byte->byte->byte->pexlif
:
```

A few comments. First, as one can see, the input binder allows multiple signals (of the same type) to be declared at the same time. Second, the “name” given as first argument to the CELL function is used in visualization of the module. We will return to that later. In general, if the name starts with ”draw_”, it is used by the visualization system (with an associated tcl/tk drawing function).

Of course, the above code created a circuit to add 9 byte sized inputs. If we instead define the functions as:

```
File                    Interrupt                    Help
: let add3 =
    input      a.
    input      b.
    input      c.
    output     o.
    internal   m.
    CELL "draw_hier add3" [
        m <- a '+' b,
        o <- m '+' c
];
add3::[hw_values(5),hw_destr(18),+(3),hw_mk_var(8),hw_constr(7),hw_size(5),hw_is
_arithmetic(2),hw_type_name(2)] *->*->*->*->pexlif
: let add9 =
    input      i1 i2 i3 i4 i5 i6 i7 i8 i9.
    output     o.
    internal   m1 m2 m3.
    CELL "draw_hier add6" [
        add3 i1 i2 i3 m1,
        add3 i4 i5 i6 m2,
        add3 i7 i8 i9 m3,
        add3 m1 m2 m3 o
];
add9::[hw_values(33),hw_destr(100),+(21),hw_mk_var(54),hw_constr(41),hw_size(33)
,hw_is_arithmetic(8),hw_type_name(8)] *->*->*->*->*->*->*->*->*->*->pexlif
:
```

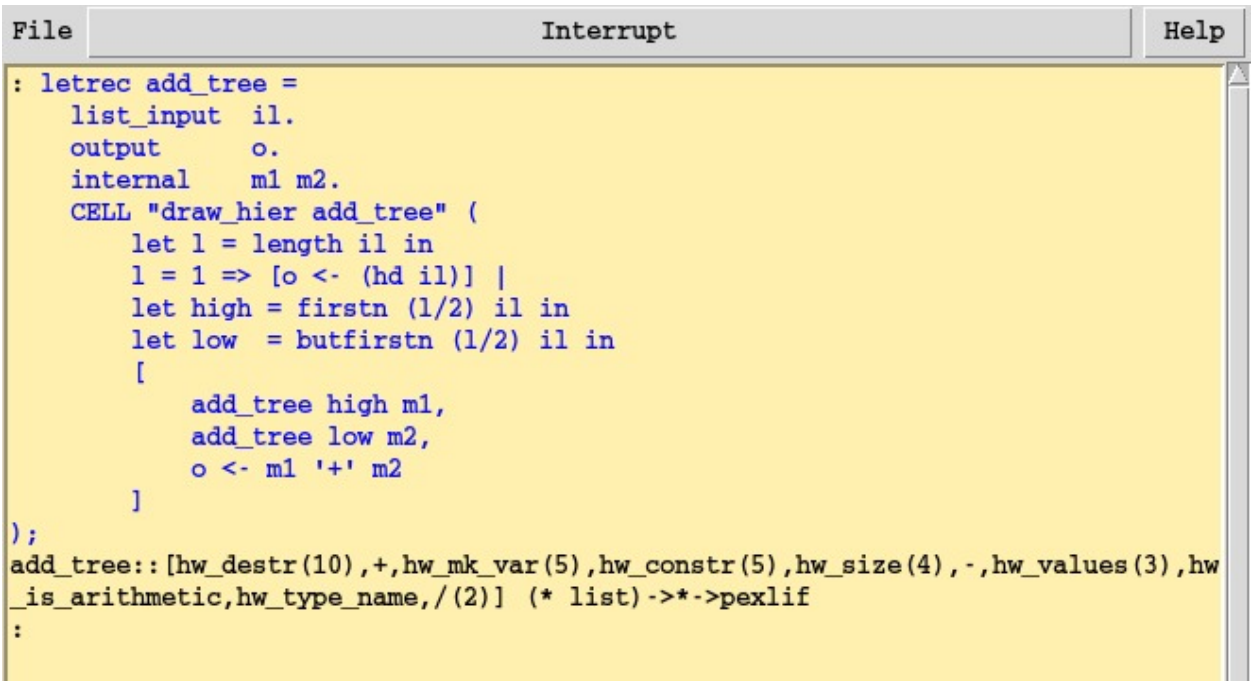
we have created a module that will add up 9 elements of any (arithmetic) type.

Of course, we may want to create a module that can be used to add up any number of inputs. The first version might look like:

```
File                    Interrupt                    Help
: letrec add_list =
    list_input  il.
    output      o.
    internal    m.
    CELL "draw_hier add_list" (
        length il = 1 => [o <- (hd il)] |
        [
            add_list (tl il) m,
            o <- (hd il) '+' m
        ]
);
add_list::[hw_destr(10),+,hw_mk_var(4),hw_constr(4),hw_size(3),-,hw_values(2),hw
_is_arithmetic,hw_type_name] (* list)->*->pexlif
:
```

Here we see that we can use standard fl constructs to recursively build up the circuit.

Of course, the last version will generate a long chain of adders which will likely be too slow in practice. Thus one would typically want to create a tree structured adder circuit so that the maximum delay will grow logratihmically with the number of terms to be added. This can be accomplised as:



2.2.4 HFL High-level Constructs

Since certain types of structures are very common in hardware, functions have been created to creating such structures. Currently there are high-level constructs for

```
CASE     -- case statements
STATE     -- simple compute state machine
SELECT     -- select a subfield based on an address
DECODER     -- decode an input vector
DECODER_LIST    -- similar to DECODER but output is list of bit signals
EN_DECODER     -- decode an input vector but with enable signal
EN_DECODER_LIST -- similar to DECODER but output is list of bit signals
Moore_FSM     -- a Moore style finite state machine
```

To illustrate the Moore_FSM primitive (which is the most complex), consider the following example:

File | Interrupt | Help

```
: // The states of the finite state machine
ENUM "my_fsm"  ["IDLE", "REQ", "MISS", "UPDATE", "HIT"];

: let fsm_example =
    bit_input       clk reset.
    bit_input       req.
    bit_input       hit.
    bit_input       valid.
    bit_output      need_data.
    bit_output      ready.
    //
    my_fsm_internal    state.
    CELL "draw_hier CACHE" [
        (Moore_FSM "CACHE_FSM" clk state
            (reset, IDLE) [
                IDLE --- req --- REQ,
                REQ  --- hit --- HIT,
                REQ  --- '~' hit --- MISS,
                MISS --- '~' valid --- MISS,
                MISS --- valid --- UPDATE,
                UPDATE --- default --- HIT,
                HIT  --- default --- IDLE
        ]),
        need_data <- is_MISS state,
        ready <- is_HIT state
];
fsm_example::bit->bit->bit->bit->bit->bit->bit->pexlif
:
```

It should be pointed out that the Moore_FSM function also creates an explicit visualization of the finite state machine that can be seen by clicking on the FSM symbol in the circuit browser. We will return to this later.

2.3 Verilog models

VossII uses the Yosys system to provide a fairly complete Verilog 2005 parser, type checker, elaborator, and lightweight synthesis path. As a result, there is an fl function, called verilog2pexlif that reads in Verilog file(s) and converts the model into a pexlif. In order to use it though, a standard library called "ste.fl" must first be loaded.

For example, if there are two files small.v and small_lib.v in the directory verilog_examples, that look like:

```
// File: small_lib.v
module  mux2(
    din_0      , // Mux first input
    din_1      , // Mux Second input
    sel        , // Select input
    mux_out      // Mux output
);
//-----------Input Ports---------------
input din_0, din_1, sel ;
//-----------Output Ports---------------
output mux_out;
//------------Internal Variables--------
reg  mux_out;
//-------------Code Starts Here---------
always @*
begin : MUX
 case(sel )
    1'b0 : mux_out = din_0;
    1'b1 : mux_out = din_1;
 endcase
end
endmodule
```

and

```
// File: small.v
module  mux4(
    din_0      , // Mux first input
    din_1      , // Mux Second input
    din_2      , // Mux Thirsd input
    din_3      , // Mux Fourth input
    sel        , // Select input
    mux_out      // Mux output
);
//-----------Input Ports---------------
input din_0, din_1, din_2, din_3 ;
input [1:0] sel ;
//-----------Output Ports---------------
output mux_out;
//------------Internal Variables--------
reg  mux_out;
reg  mid01, mid23;
//-------------Code Starts Here---------
    mux2 mux1  (.din_0(din_0), .din_1(din_1), .sel(sel[0]), .mux_out(mid01));
    mux2 mux2  (.din_0(din_2), .din_1(din_3), .sel(sel[0]), .mux_out(mid23));
    mux2 mux12 (.din_0(mid01), .din_1(mid23), .sel(sel[1]), .mux_out(mux_out));

endmodule
```

We could now do:

```
File                          Interrupt                          Help
: load "ste.fl";
-Loading file /home/cseger/VossII/vosslib/ste.fl:
T
it::bool
: let p = verilog2pexlif "-Iverilog_examples"          // Yosys flags
                         "mux4"                       // Top-level module
                          ["small.v", "small_lib.v"]  // Files to read & compile
                          []                          // Additional files needed
;
p::pexlif
:
```

followed by:

```
File                          Interrupt                          Help
: number_of_wires p;
-Loading file /tmp/voss2_cseger_o4I89F/mux4_verilog_bkl5eW/_out.fl:
: 5
it::int
:
```

or

```
File                    Interrupt                    Help
: extract_subckts "not" p;
[:
(PINST "draw_not" [src->/home/cseger/VossII/IDVII/src/fl/design/yosys_lib.fl:26,
src->small_lib.v:23|small_lib.v:22] T
  [(A,[sel])]
  [(Y,[TmP_0])]
  []
  LEAF [
    Y <- ~(A)
  ]
)
,:
(PINST "draw_not" [src->/home/cseger/VossII/IDVII/src/fl/design/yosys_lib.fl:26,
src->small_lib.v:23|small_lib.v:22] T
  [(A,[sel])]
  [(Y,[TmP_0])]
  []
  LEAF [
    Y <- ~(A)
  ]
)
,:
(PINST "draw_not" [src->/home/cseger/VossII/IDVII/src/fl/design/yosys_lib.fl:26,
src->small_lib.v:23|small_lib.v:22] T
  [(A,[sel])]
  [(Y,[TmP_0])]
  []
  LEAF [
    Y <- ~(A)
  ]
)
]
it::pexlif list
:
```

2.4 fsm

Although the pexlif format is convenient for representing a circuit, it is not very efficient for simulation. Thus, an internal datatype, called fsm, is used. A pexlif can be converted to an fsm by simply calling pexlif2fsm.

```
File | Interrupt | Help
: let ckt = pexlif2fsm p;
ckt::fsm
: ckt;
FSM for draw_hier mux4 with 18 nodes and 12 composites
fsm
it::fsm
:
```

There a number of functions that operate on the fsm datatype. For example:

```
File | Interrupt | Help
: // Get a list of all inputs to the circuit
inputs ckt;
["din_0","din_1","din_2","din_3","sel[1:0]"]
it::string list
: // Get a list of all outputs of the circuit
outputs ckt;
["mux_out"]
it::string list
: // Return the immediate fanin from a node in a circuit.
fanin ckt "mux_out";
["i2/TmP_0","mid01","i2/i2/__tmp1"]
it::string list
: // Return the immediate fanout from a node in a circuit.
fanout ckt "mid01";
["mux_out"]
it::string list
: // Find the full vector given a node in it.
node2vector ckt "sel[0]";
"sel[1:0]"
it::string
: // Use fanin_dfs to recursively go back until a toplevel node is encounterd
let top_fanin ckt nd =
    let toplevel n = NOT (str_is_substr "/" n) in
    let all_inps = fanin_dfs ckt toplevel [nd] in
    let top_level_inps = filter toplevel all_inps in
    top_level_inps subtract [nd]
;
top_fanin::fsm->string->(string list)
: top_fanin ckt "mux_out";
["mid23","sel[1]","mid01"]
it::string list
:
```

As usual, use the help system to find all functions that can be applied to an fsm.

3 Symbolic Trajectory Evaluation and Verification

There is currently only one built-in command for symbolic trajectory evaluation called STE. In general, STE determines, through symbolic trajectory evaluation, whether an antecedent/consequent pair hold in for some circuit. STE will return a datatype called ste which contains all the result from the STE run. This includes not only the result of the STE run, but also trace data etc. collected during the simulation.

In general STE is invoked as

```
STE options ckt wl ant cons trl
```

where options is a string that can contain a combinations of the following flags:

-a Abort the verification at the first antecedent or consequent failure.

-m n Abort the verification after n phases has been simulated.

-e Evaluate evey node in the circuit whether the node is in the fan-in tree of some node that is checked or traced or not. In other words, with the -e option, the simulator will compute the value on every node. Normally, a node whose value cannot be observed (directly or indirectly) is simply kept at X. Of course, the STE command can run significantly slower with the -e option, so beware!

The second argument to STE must be an object of the fsm type representing a circuit that is to be simulated.

The weak_list wl is a list of 4-tuples. Each 4-tuple is of the form (g,n,f,t), where n is a node name, f and t denote start and stop times, and g is a Boolean denoting the domain for which this node should be weakened. For assignments making the guard true, the next state function of the circuit node is set to X. This is often useful when verifying a subsystem of a larger system that has shared input signals. By weakening the model, unnecessary computations do not have to be performed. Note that weakening a model is a "safe" operation in that the monotonicity of the circuit models guarantee that if we can verify something in the weakened model, then that same property is guaranteed to hold in the original model.

The ant_list and cons_list are both lists of five-tuples. Each five-tuple is of the form (g, n, v, s, t), where g is a Boolean function denoting the domain for which this assertion/check should be carried out, n is the name of a node, v is the value to be asserted/checked, and s and t denote the start and stop times for this assertion/check respectively.

Finally, the last argument to STE is a list of triples. Each triple is of the form (n, s, t), where n is a name of a node to be traced and s and t are the start and stop times for this trace respectively.

Of course, in practice, it would be quite tedious to have to write all specifications in terms of lists of five-tuples. Consequently, a small embedded domain specific language has been defined. These functions make it much easier to write specification. However, it should be remembered that when the verification is actually performed, all these higher level constructs gets translated down to the two lists of five-tuples. The DSL has the grammar:

```
antecedent    : trace 'and' antecedent
              | trace

trace         : <node_name> is values
              | <vector_name> is values

values        : time_sequence
              | time_sequence 'otherwise' value 'until' duration

time_sequence : timed_value
              | timed_value 'followed_by' time_sequence

timed_value   : value 'for' duration
              | value 'in_phase' <integer>
              | value 'in_cycle' <integer>

duration      : <integer> 'phase'
              | <integer> 'phases'
              | <integer> 'cycle'
              | <integer> 'cycles'

value         : "0x..."     // Hex constant
              | "0b..."     // Binary constant
              | "<integer>" // Decimal constant
              | "[a-zA-Z]..."       // Variable, e.g., a[3:0], b[1:0][2:0]
              | <bv>                // Any bv of suitable size (or smaller)
              | <bool list> // Of the same length as vector
```

For more details on how to use the STE function to perform formal verification, we refer the reader to Section **??**.

To illustrate the use of STE on a simple example, consider the simple Verilog circuit from the previous section. In this example, we will simply use STE as a symbolic simulator.

```
File                          Interrupt                          Help
: load "ste.fl";
-Loading file /home/cseger/VossII/vosslib/ste.fl:
-Loading file /home/cseger/VossII/IDVII/src/fl/design/STE.fl:
-Loading file /home/cseger/VossII/IDVII/src/fl/design/util.fl:


-Loading file /home/cseger/VossII/vosslib/gen_u_arithm.fl:
T
-Loading file /home/cseger/VossII/IDVII/src/fl/design/hfl.fl:


-Loading file /home/cseger/VossII/IDVII/src/fl/design/yosys_lib.fl:
-Loading file /home/cseger/VossII/IDVII/src/fl/design/read_verilog.fl:
-Loading file /home/cseger/VossII/IDVII/src/fl/gui/sch_draw/types.fl:
-Loading file /home/cseger/VossII/IDVII/src/fl/gui/sch_draw/draw_sch.fl:
""
T
it::bool
: let p = verilog2pexlif "-Iverilog_examples"         // Yosys flags
                        "mux4"                       // Top-level module
                         ["small.v", "small_lib.v"]  // Files to read & compile
                         []                          // Additional files needed
;
p::pexlif
: let ckt = pexlif2fsm p;
ckt::fsm
:
```

If we now define a simple antecedent, as follows:

```
File                          Interrupt                          Help
: let ant =
    "din_0" is  "a" for 5 phases
  and
    "din_1" is 0 for 1 phase followed_by "0x1" for 2 phases followed_by
               0 for 1 phase followed_by "b" for 1 phase
  and
    "din_2" is "c" for 5 phases
  and
    "din_3" is "d" for 5 phases
  and
    "sel[1:0]" is   "0x0" for 1 phase followed_by
                    "0x1" for 1 phase followed_by
                    "0x2" for 1 phase followed_by
                    "0x3" for 1 phase followed_by
                    "s[1:0]" for 1 phase
;
ant::bool#(string#(bool#(int#int))) list
:
```

we can run STE

```
File                    Interrupt                    Help
: let ste_res = STE "-e" ckt [] ant [] [];
ste_res::ste
: ste_res;
-Loading file /tmp/voss2_cseger_ZuvHV4/mux4_verilog_5waIiA/_out.fl:
: Time: 0
Time: 1
Time: 2
Time: 3
Time: 4
Time: 5
Time: 6
STE result for circuit draw_hier mux4 simulated to time 6
ste
it::ste
:
```

and potentially see the output values on mux_out:

```
File                    Interrupt                    Help
: get_trace ste_res "mux_out";
[(5, (T, T)),(4, (a&b&c&d + c&s[1]&s[0]' + a&b&s[1]' + d&s[1]&s[0] OR ... , d'&s
[1]&s[0] + c'&s[1]&s[0]' + b'&s[1]'&s[0] OR ... )),(3, (d, d')),(2, (c, c')),(1,
 (T, F)),(0, (a, a'))]
it::int#(bool#bool) list
: // See it in increasing time instead.
rev it;
[(0, (a, a')),(1, (T, F)),(2, (c, c')),(3, (d, d')),(4, (a&b&c&d + c&s[1]&s[0]'
+ a&b&s[1]' + d&s[1]&s[0] OR ... , d'&s[1]&s[0] + c'&s[1]&s[0]' + b'&s[1]'&s[0]
OR ... )),(5, (T, T))]
it::int#(bool#bool) list
:
```

Note that get_trace returns a list of triples: (t, H, L), where t is the phase the node took on this value and H and L are the high and low rail values of the node. The encoding used is (T, T) is X (unknown/should't matter), (T, F) is one, (F, T) is zero, and (F, F) is top, or overconstrained.

Although the above interface can be used, **VossII** comes with a GUI for visualize a circuit and the result of an STE run. To invoke this debugger, issue the command `STE_debug` with the fsm as argument.

```
File                    Interrupt                    Help
: let vis = STE_debug ckt;
vis::vfsm
:
```

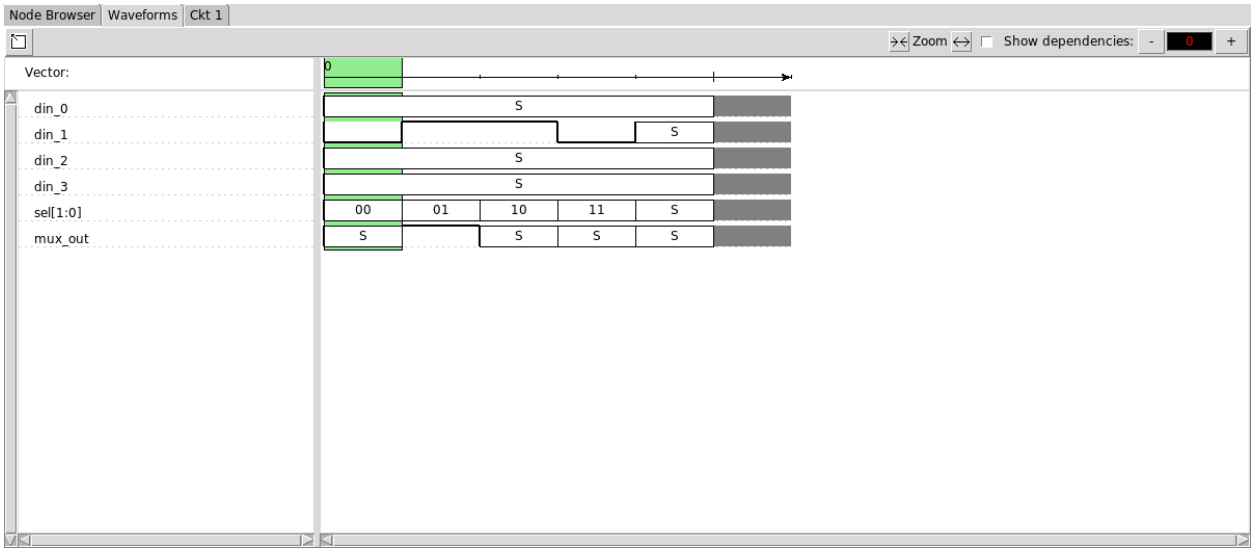
If `vis` is evaluated a new toplevel window will be opened looking like:

From here, you can select some nodes (searchable) and ask to see their fanin cone. For example, if you select the `mux_out` signal and click on the Fanin button, you get:

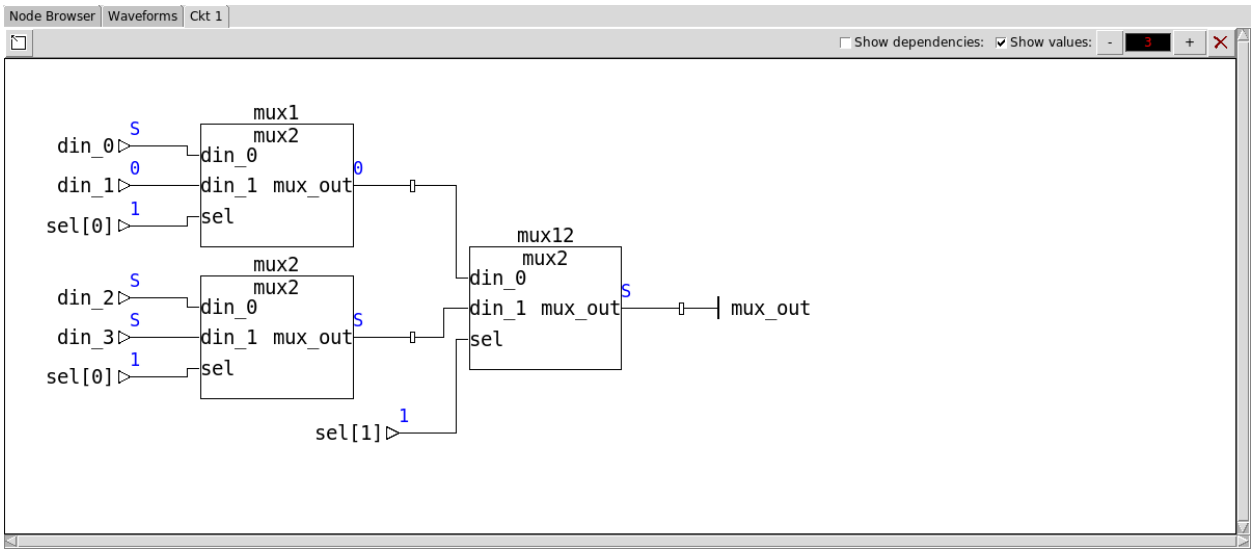
If you now run STE, but instead of giving ckt, give it vis (STE is overloaded), you can see the values in the GUI.

```
File                    Interrupt                    Help
: STE "-e" vis [] ant [] [];
Time: 0
Time: 1
Time: 2
Time: 3
Time: 4
Time: 5
Time: 6
STE result for circuit draw_hier mux4 simulated to time 6
ste
it::ste
:
```

If you now add all the inputs and the output to the Waveforms window (by selecting them and clicking the Waveform button), you get (after zooming out a bit):



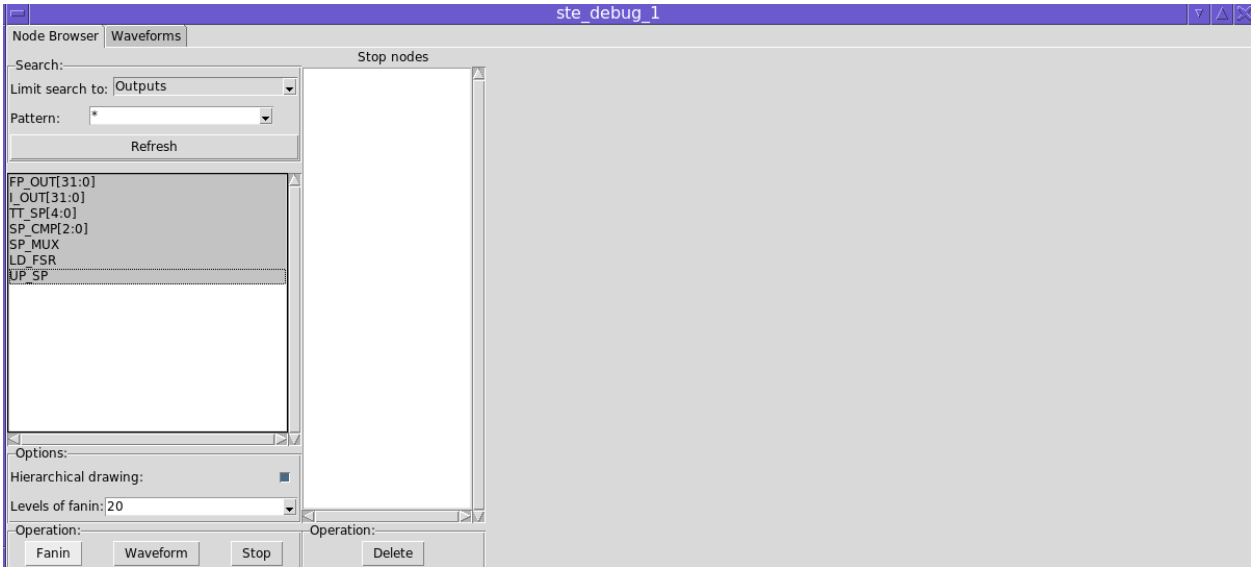
By clicking on the time line, you can set the "current" time point and if you then go back to the fanin viewer and selecting "Show values" you get:



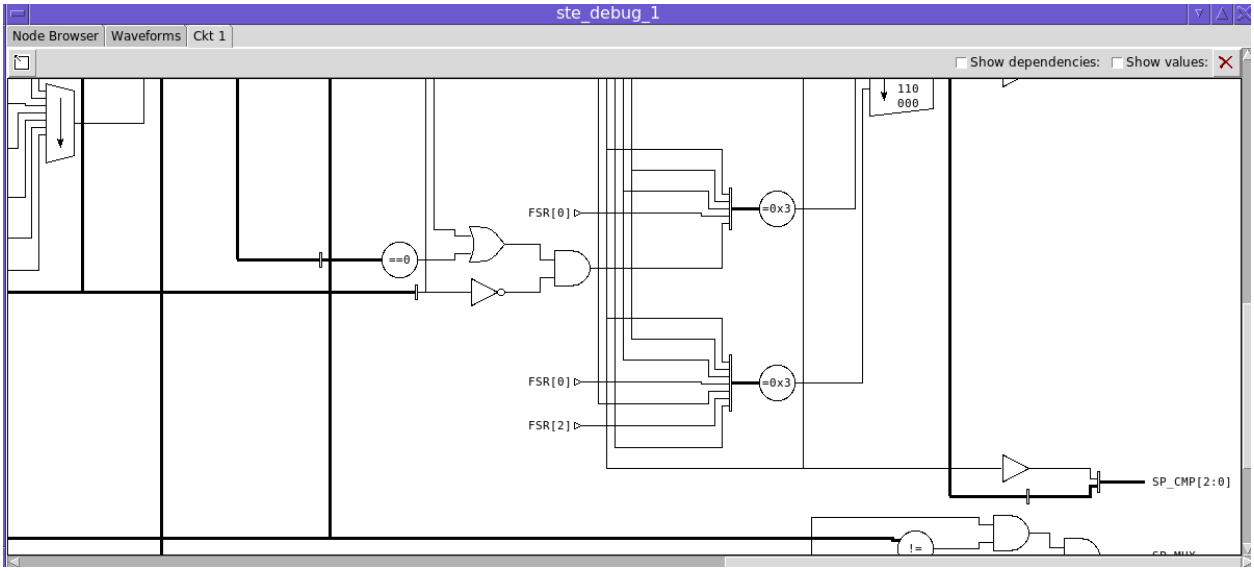
To see what the symbolic value "S" in the viewer really represent, hover over the S or right click on S. By clicking on the right or left arrows, you can change the time point. Finally, any of the windows in the GUI can be detached (except the first one) and later re-attached by clicking on the "detach" icon at the top-left corner of each pane.

3.1 GUI Commands

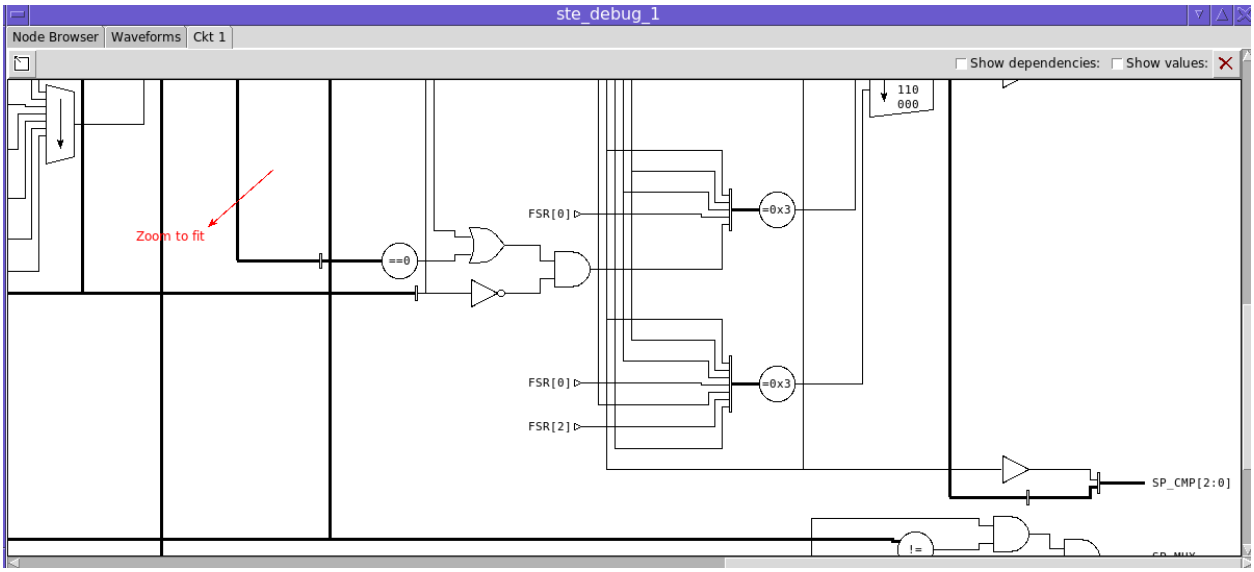
To illustrate the use of the GUI and explain many commands, we will use the single precision floating point unit in `<VossII installation directory>/ckt_examples/m32632/rtl`. If you want to follow along, cd into that directory and execute `fl -f SP_FPU.fl`. If you do so, you will first be greeted by:



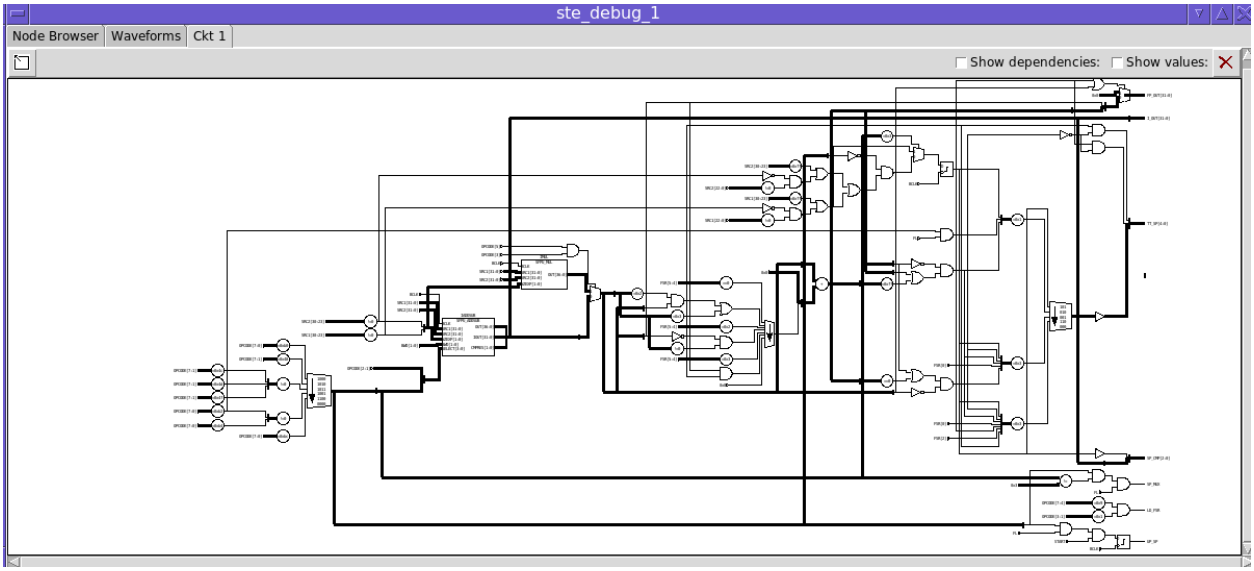
You can now select one of more of the signals shown. If the signal(s) you are interested in are not primary outputs, change the Limit search, provide a pattern and click Refresh. Here we selected all of the primary outputs and clicked the Fanin button obtaining:



If you now click and hold down the right button while moving the cursor down and left, a red arrow and a "Zoom to fit" text appears.

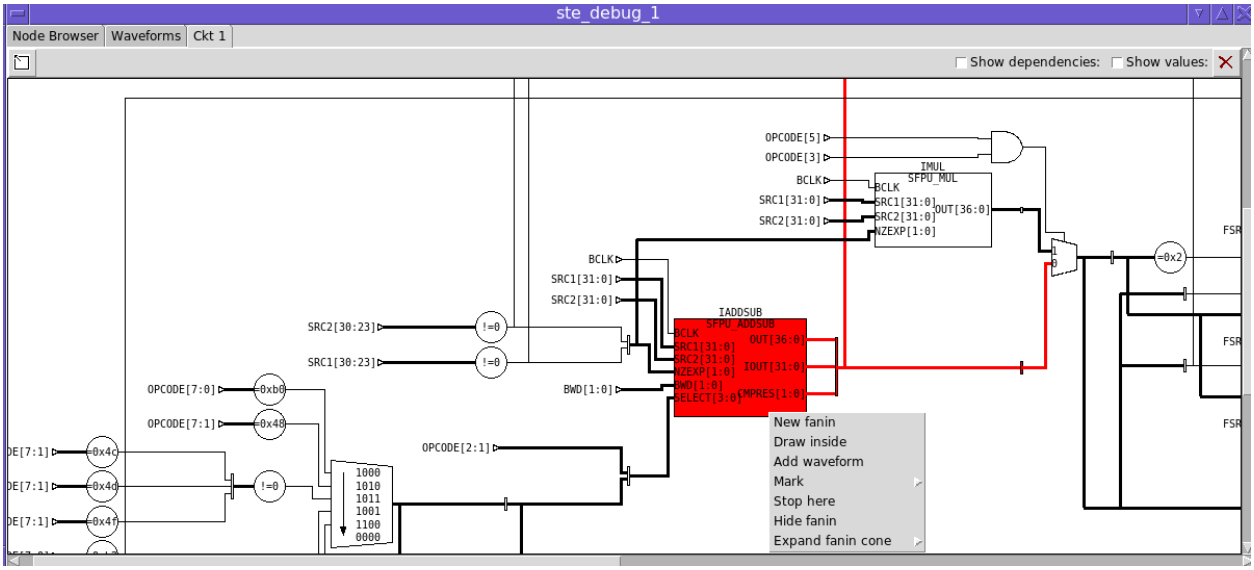


You can also strike up-left, up-right, or down-right while holding the right button down for other zoom alternatives. If you use the Zoom to fit alternative and release the button, you get:

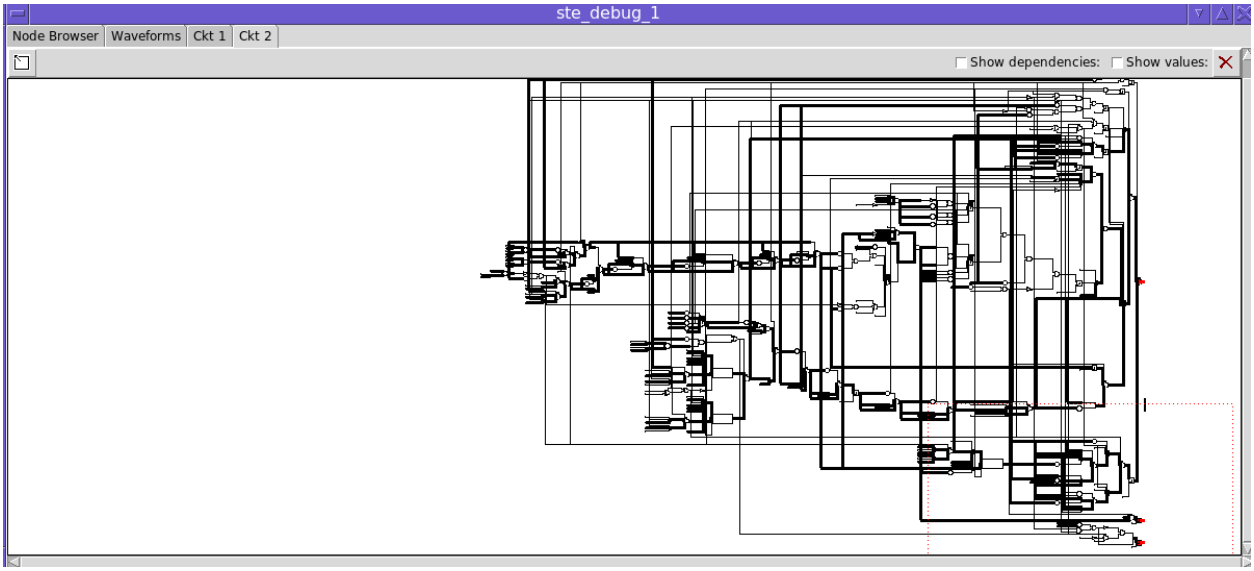


Of you zoom in some (mouse wheel can also be used), you can see that hover over a wire or box displays the name of the wire or output(s) of box.

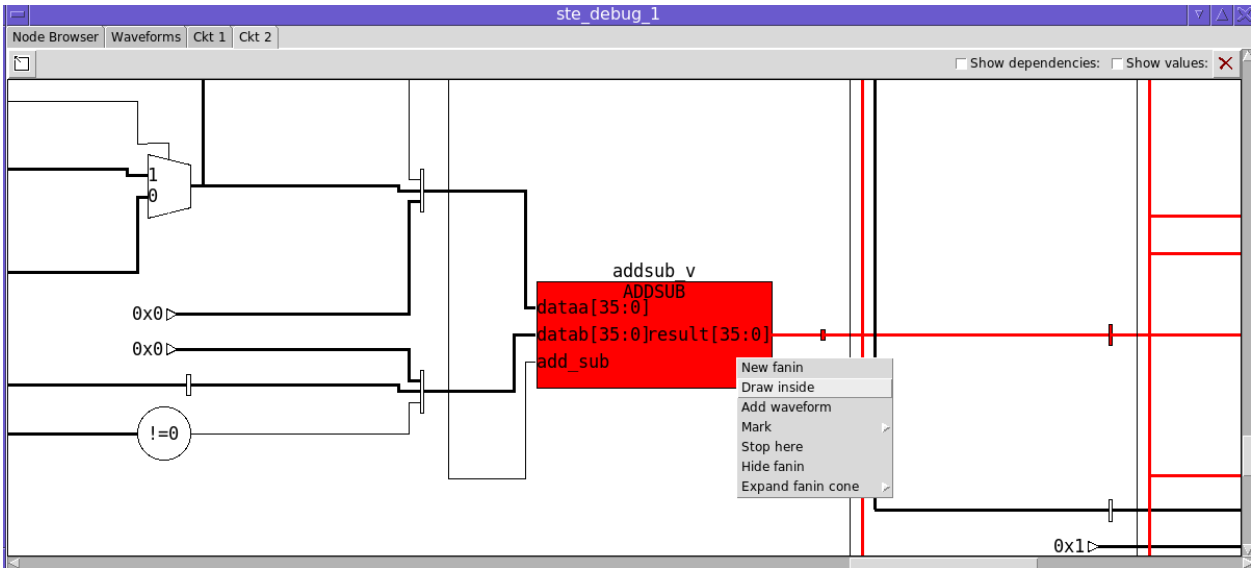
If you right click on a wire or box, a pulldown menu appears:



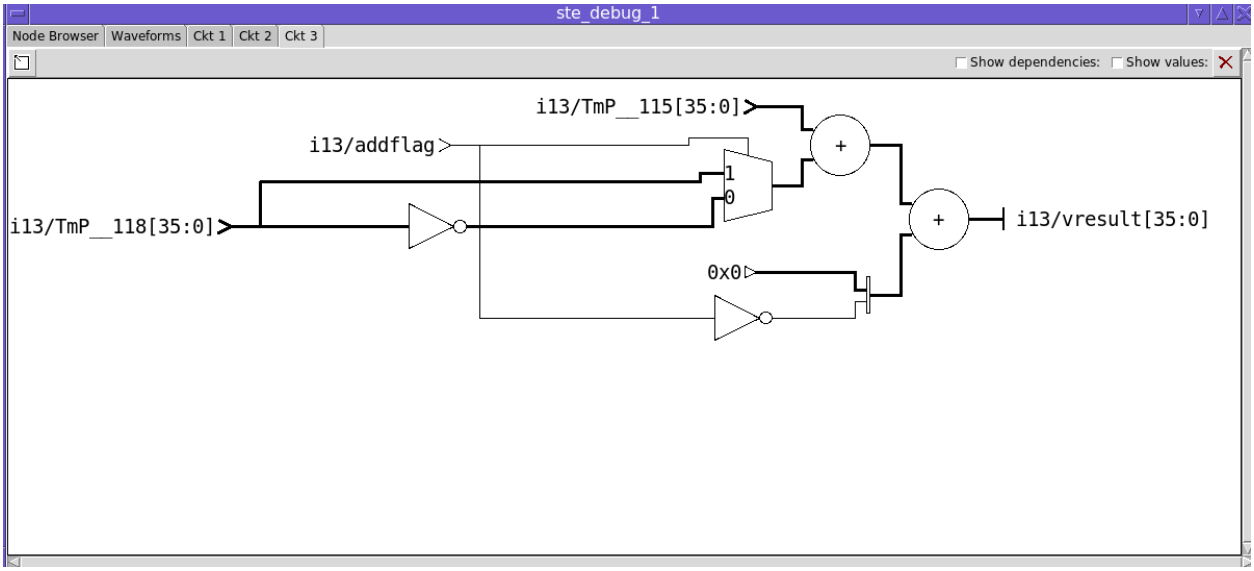
Here we selected the “Draw inside” alternative and a new tab gets created and we get to see the inside circuitry of that box.



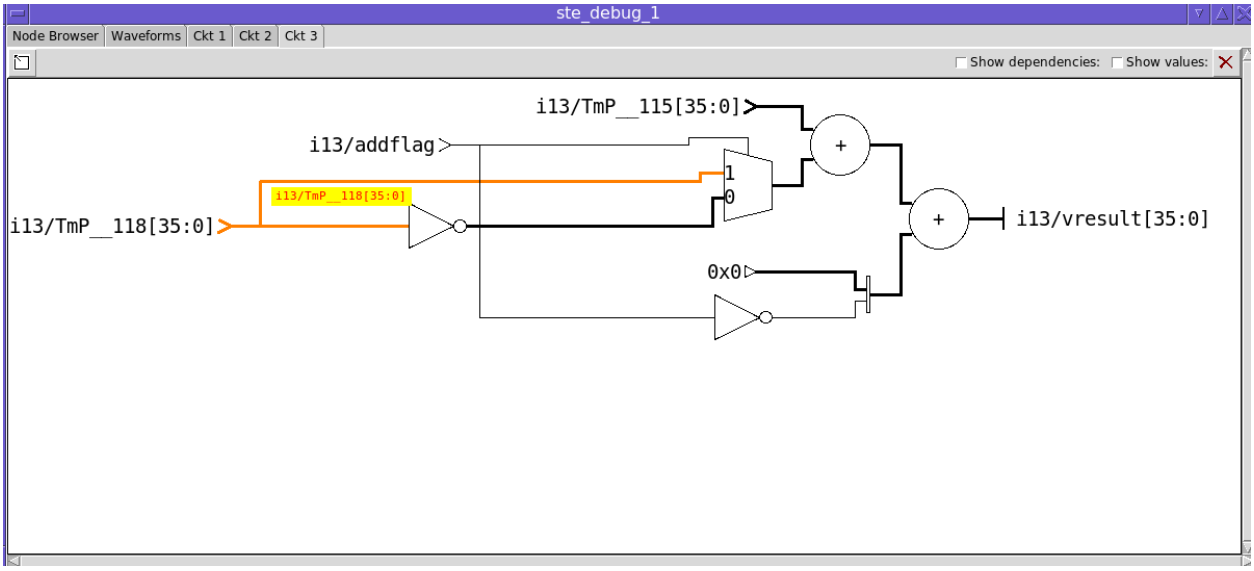
If we do the same inside this level of hierarchy, like:



we get:

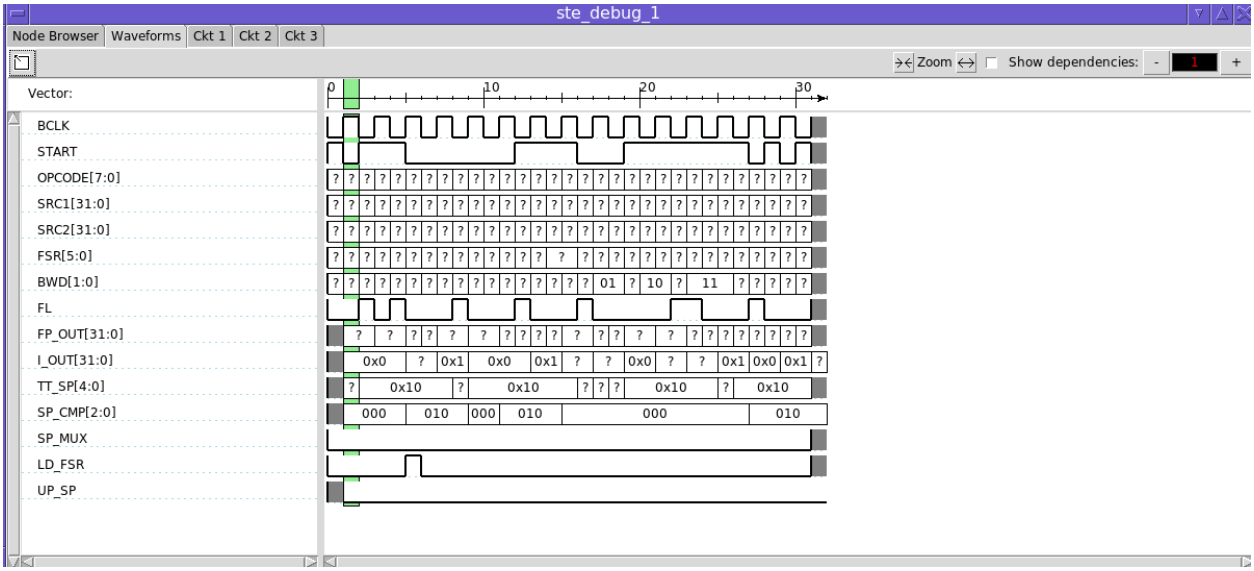


Since it is easy to get lost when many levels of hierarchy have been traversed, it is often very useful to color a wire. One can do this either by the pulldown menu described earlier, or simply hit a digit between 1 and 9 on the keyboard.



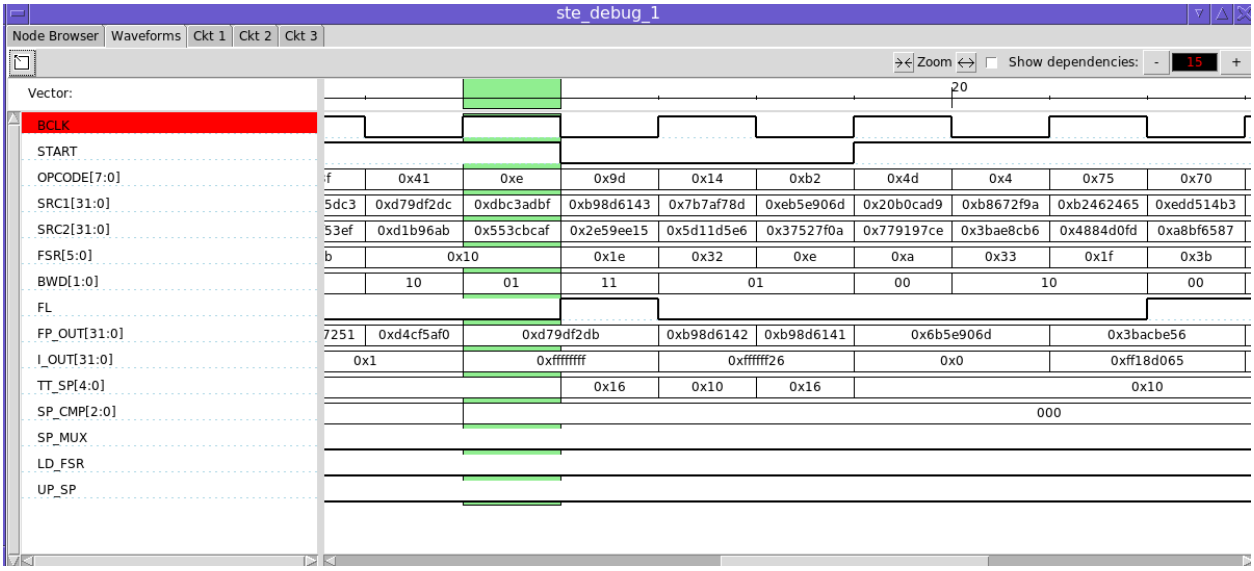
The important part is that the selected wire will be colored in the chosen color in every window in which it occures (including the waveform window).

As we illustrated earlier, one can also select wires to add to the waveform viewer. Here we fist selected all inputs followed by all the output after having run a (semi-) random scalar input simulation.

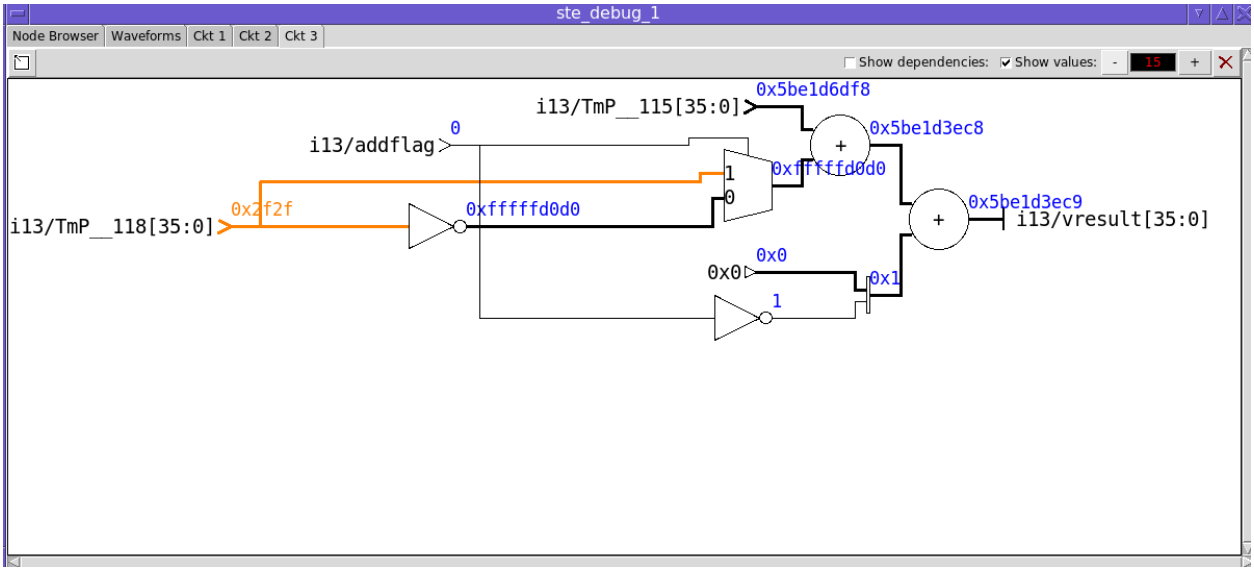


If there are question marks in the drawing, it means that the value did not fit in the space. Hovering over it will often provide the value or right clicking to get a pulldown menu and then select the "Show expanded value" menu item. An alternative is to zoom out so that enough room is created.

By clicking on the time line, you can set the current time point, which will be reflected in every window for which "Show values" has been selected. For example:

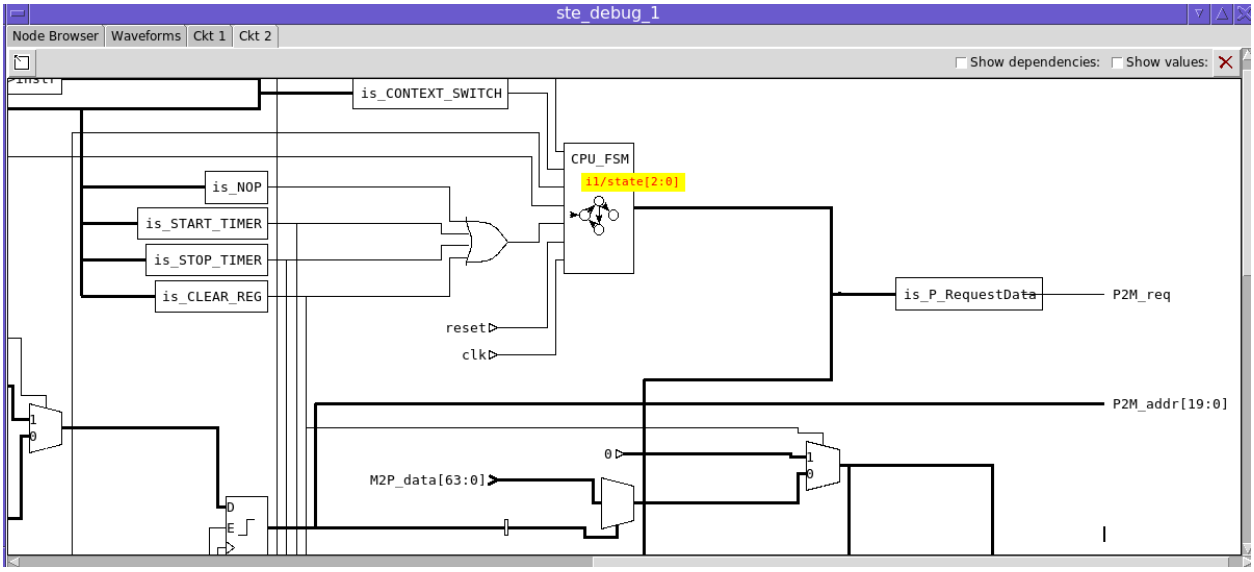


and

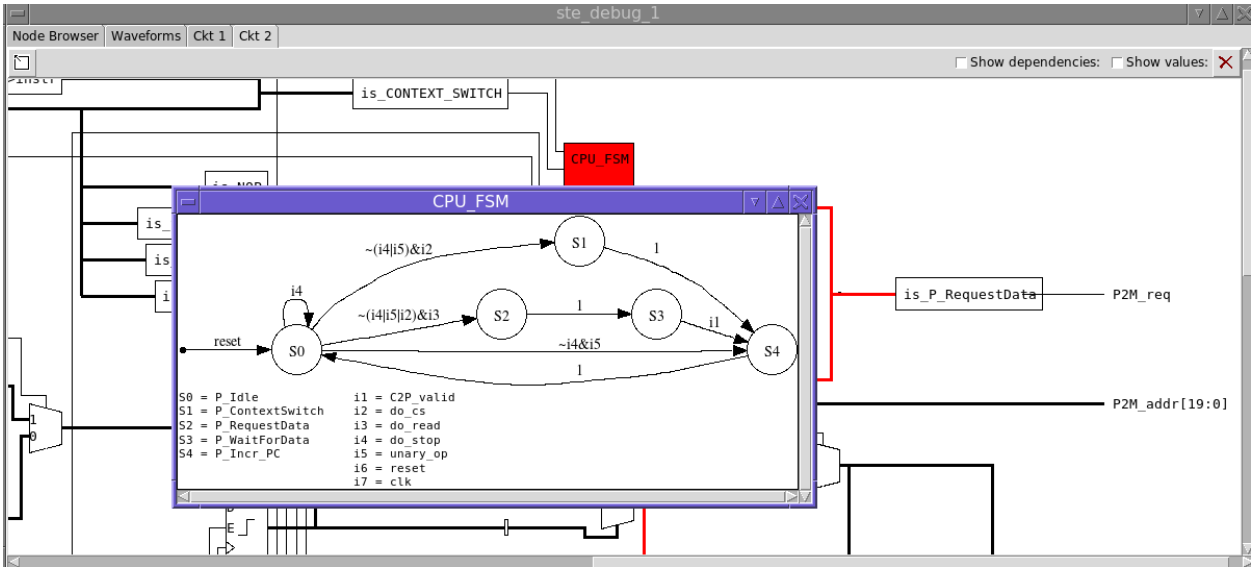


If you want to re-order the signals in the waveform viewer, place the cursor over the signal you want to move. A small green arrow will appear. You can now use the up or down arrows to move this signal. Similarly, if you hit the Delete key, that signal (with its waveform) will be removed from the waveform viewer.

Our final example of GUI features is using a different circuit. Now, cd into `<VossII installation directory>` and issue the command `fl -f ex3.fl`. This is a circuit defined in HFL and thus there are additional visualization features possible. If we select all outputs and click Fanin and then dive into the CPU hierarchy, we get:



Notice that the finite state machine (defined using the Moore_FSM construct in HFL), is drawn specially. If you now left click on the symbol, you get an explicit representation of the finite state machine:



where you can hover over states and/or transitions to get the original information from HFL displayed.

Furthermore, if have run an STE simulation and you now move the time line, the state the machine is in will be highlighted in dark green. If the state is partially X or symbolic, every state that it could be in will be colored light green.

Finally, if the state is scalar, both the back annotation value shown for the output of the state machine as well as the value in the waveform viewer will use the name of the state, rather than the encoded value. All to make debugging easier!

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] Michie, D. ""Memo Functions and Machine Learning.", *Nature*, Vol 218, (1968), pp. 1922,

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

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

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] Srensson, N, and En, N. "MiniSat 2.1 and MiniSat++", 1.0SAT race 2008 editions. SAT (2009): 31.

[42] 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.

[43] 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