incOSHL User Manual and Developer Guide

Hao Xu
xuh@cs.unc.edu

September 2, 2012

1 User Manual

1.1 Compiling and Linking the Source Code

All source code of incOSHL is located in one directory. To compile to the source code, go to the directory contain the source code and run the following command:

```
g++ -O3 -o incOSHL *.cpp
```

The g++ version used for development is 4.6.3. And the g++ version used for testing is 4.1.2. Other version of g++ may also work, but they are not tested.

1.2 Running the Prover

To run incOSHL, use the following command:

```
incOSHL <file_path> <options>
```

<file_path> is a relative path to the tptp path, which can be specified in <options>. <options> is a comma separated list of key value pairs of the form <key>=<value>. Most supported keys include both a long form and a short form. The supported keys are given in the following table.

Key Long Form	SF	Value Type	Usage

Key Long Form	SF	Value Type	Usage
timeout	to	integer (seconds)	This key sets the cpu clock time limit for the main loop. The prover periodically checks if the cpu time has exceeded this limit and stops the proof search. In some case. The periodical check may not occur around the time limit, resulting a longer running time. The "timeout" command from coreutils can be used to solve this problem.
useSuperSymbol	uss	boolean	This key sets whether the prover creates supersymbols during the preprocessing phase. When enabled, it replaces multiple consecutive symbols in a flatterm with a supersymbol, which provides minor efficiency improvement on some problems.

Key Long Form	SF	Value Type	Usage
sortSubtreesBySize	ssbs	boolean	This key sets whether the prover sorts the subtrees of every clause tree node by the size of the minimum clause instance that can be generated from those subtrees, in ascending order of the minimum clause instance size of a subtree, a value computed during preprocessing. When enabled, it allows the prover to stop looking further at other subtrees when the minimum clause instance size of a subtree is larger than the current term size limit.
stacksize	ss	integer (megabytes)	This key sets the memory size limit for the prover. The prover generates a "max memory" error if the memory usage is greater than this size plus a small constant (non-growing) memory footprint. In some case, memory leak causes the prover go over this memory usage limit.
flipLiteral	fl	boolean	This key sets whether the sign of the literals should be flipped.
randomFlipLiteral	rfl	boolean	This key sets whether the prover randomly chooses a subset of predicates and randomly flips the signs of all the literals with predicates in that set.

Key Long Form	SF	Value Type	Usage
randomLexicalOrder	rlo	boolean	This key sets whether the prover randomly assigns the lexical order of symbols, as opposed to the order they appear in the input file.
randomClauseInsert	rci	boolean	This key sets whether the prover randomly assigns the order in which the input clauses are inserted into the clause tree, as opposed to the order they appear in the input file.
randomClauseLiteral	rcl	boolean	This key sets whether the prover randomly assigns the order in which literals appear in an input clause when the clause is inserted into the clause tree, as opposed to the order they appear in that clause in the input file.
tptppath	tp	string (with quotes)	This key sets the tptp path.

1.3 Prover Output

The prover generates on of the following outputs:

output	
time out	The prover timed out without finding a proof.
max memory	The prover reached the memory limit without finding a proof.
max term size	The prover reached the maximum term size without finding a proof.
proof found	The prover found a proof.
saturated	The prover found a model.

1.4 Compiling Options

There are a few macro-based switches that changes the behavior of the prover. They are defined in "defs.h". Some of these switches are for debugging or profiling purposes, and I have not always been updating them to match the current

source code. They may not work in the current code. Even if they work, but they may not provide enough information without further tweaking. The following table lists the macros:

Macro	Usage
INCREMENTAL_GENERATION	Generate instances incrementally.
USE_TYPES	Use types.
USE_RELEVANCE	Use relevance.
STACK_DEBUG	Print stack information.
STACK_STATE_DEBUG	Print program state information.
PROFILE_MODEL	Profile the model update time.
DEBUG_LOOKUP	Print information generated by the lookup coroutine.
DEBUG_INVERSE_LOOKUP	Print information generated by the inverse lookup coroutine.
DEBUG_INSTANCE	Print the generated contradicting instance.
DEBUG_INSTANCE_CLAUSE	Print the input clause that generated the contradicting instance and the substitution.
DEBUG_ORDERED_RESOLVE	Print information generated the ordered resolution.
DEBUG_ELIGIBLE_LITERALS	Print eligible literals and the instances that generated those eligible literals
DEBUG_RESTART_SIZE	Print the restart size for each new clause instance search. The restart size is the lower bound of the size of possible contradicting clause instances, calculated based on the changes in the model. The basic idea is that if a change in the model only affected (add or remove) eligible literals of size X and higher and we have searched all possible instances of size X-1 or lower before this change, then after this change, we can restart our search for a contradicting clause instance from size X.
DEBUG_CLOCK	Print the current clock value at various places in the code.
DEBUG_INSTANCE_CLOCK	Print the current clock value after generating a contradicting clause instance.
DEBUG_TERM_SIZE_LIMIT	Print the universal term size limit.

Macro	Usage
DEBUG_LOOKUP_CLAUSE_TREE	Print information generated by the clause tree traversal coroutines.
DEBUG_GENERATE_TERM	Print information related to term generation in the reverse lookup coroutine.
DEBUG_VAR_TYPE	Print the type of a free variable when the reverse lookup coroutine tries to generate terms for it.
DEBUG_TRIE	Print the model trie and changes on the trie.
DEBUG_MODEL	Print information about the changes on the model.
DEBUG_RESTART	Print information about garbage collector.
DEBUG_DISABLE_ENABLE	Print information about global mutable data dependency.
DEBUG_VERIFY_GENERATED_INSTANCE	Verify if the generated instance is correct.
DEBUG_VERIFY_RESOLVENT	Verify if the resolvent is correct.
DEBUG_VERIFY_EL_CLAUSES	Verify if the eligible literals are correct.
DEBUG_CHECK_THREADS	Check the integrity of threads.

2 Developer's Guide

2.1 macros.h

The macros.h provides the essential functions for compile-time transformation of source code, based on C/C++ macro. It is the foundation of stack.h, which implements the STACK EL. In this section, we list the basic compile-time macro functions and data structure that are defined in macros.h.

2.1.1 Data Structures

Compile-time data structures are source code. There are only three types of compile-time data structure used.

Type 0: Code fragments (CFs) without top level commas.

Type 1: This type of data structures consists of a list of code fragments (CFs) of the form:

```
1 (<cf 1>, <cf 2>, ..., <cf n>)
```

For example, one can write:

```
(1, 2, ..., 3)
```

Type 2: This type of data structures consists of a list of code fragments (CFs) of the form:

```
(<cf 0>, <cf 1>, ..., <cf n-1>, EOL)
```

For example, one can write:

```
(1, 2, ..., 3, EOL)
```

The only difference between the Type 1 and Type 2 is that the latter has a end of list marker EOL. The end of list marker is useful only for compile-time transformation and should never appear in the generated code. Therefore, we do not define its runtime behavior, having the EOL marker in the generated code results in a undefined token compiler error.

2.1.2 Type 0 Functions

Predecessor:

```
PREC(<n>) = <n - 1>
```

where $1 \leq n \leq 20$.

Successor:

```
SUCC(<n>) = <n + 1>
```

where $0 \leq n \leq 19$.

2.1.3 Type 1 Functions

Test if a list is empty:

```
IS_EMPTY_NO_EOL(()) = 1
IS_EMPTY_NO_EOL((<cf 0>, ..., <cf n-1>)) = 1
```

where $1 \leq n \leq 20$.

Append EOL:

```
APPEND_EOL((<cf 0>, ..., <cf n-1>)) = (<cf 0>, ..., <
    cf n-1>, EOL)
```

Expand:

```
EXPAND((<cf 0>, ..., <cf n-1>)) = <cf 0>, ..., <cf n
    -1>
```

List:

```
LIST(<cf 0>, ..., <cf n-1>) = (<cf 0>, ..., <cf n-1>,
    EOL)
```

2.1.4 Type 2 Functions

Get an element from a list:

```
GET(i, (<cf 0>, ..., <cf n-1>, EOL)) = <cf i>
```

where $1 \leq n \leq 20$ and $0 \leq i < n$.

Count the number of elements in a list:

```
COUNT((<cf 0>, ..., <cf n-1>, EOL)) = n
```

where $0 \leq n \leq 20$.

Get the first element of a list:

```
HEAD((<cf 0>, ..., <cf n-1>, EOL)) = <cf 0>
```

where $1 \leq n \leq 20$.

Get the tail of a list[1]:

```
TAIL((<cf 0>, ..., <cf n-1>, EOL)) = (<cf 1>, ..., <cf
    n-1>, EOL)
```

where $n \geq 1$.

Test if a list is empty:

```
IS_EMPTY((EOL)) = 1
IS_EMPTY((<cf 0>, ..., <cf n-1>, EOL)) = 0
```

where $1 \leq n \leq 20$.

MAP_FOLDR: This function is the basis for most complex functions, we will use a notation similar to function programming langauges to describe its function. The particular design here, which combines the map and foldr functions, is due to the limitation of C/C++ macros. The size of lists that it can process is generally limited to below 20.

```
map_foldr(nil, one, cons, farg, f, arg, list) =
    match list with
        () => nil
        (x) => one(nil, cons, f(x, arg))
        x::y => cons(f(x, arg), map_foldr(nil, one,
            cons, farg, f, farg(arg), y)
```

The following macros are defined as usual based on MAP_FOLDR:

MAP:

```
map(f, list) =
    match list with
        () => nil
        x::y => f(x)::map(f, y)
```

MAP_ARG:

[1] This function works on both Type 1 and Type 2 data structures

```
map_arg(f, arg, list) =
    match list with
        () => nil
        x::y => f(x, arg)::map_arg(y)
```

FOLDR:

```
foldr(nil, cons, list) =
    match list with
        () => nil
        x::y => cons(x, foldr(nil, cons, y)
```

For ZIP, UNZIPF, UNZIPS, REVERSE, DISTRI, PACK, DELETE_TAIL, APPEND, CONS, DELETE_EOL, FST, SND, CONCAT, ACCU, ACCU_REV, SHIFT, IS_ZERO, REPEAT, RANGE1, APPLY_N, DIFF, ROTATE_LEFT, their documentation can be found in the source code.

2.1.5 Adding Support for Larger Lists

The current macros are written in a way that they can be easily extended to support larger lists. To support lists of size larger than 20, only the following macros need to be modified:

- new __PREC<n> macros
- new __SUCC<n> macros
- extend __IS_EMPTY_NO_EOL and IS_EMPTY_NO_EOL_EXPAND_LIST
- new __GET<n> macros: this can be done by using the following generic template:

```
#define __GET<n>(PREC_REC,x1,...) \
    EVAL_PARAMS_GET##PREC_REC(PREC_REC, PREC(
        PREC_REC) , ##__VA_ARGS__)
```

- extend __COUNT and COUNT
- new __MAP_FOLDR<n> and APP_EVAL_PARAM_MAP_FOLDR<n> macros: this can be done by using the following generic template:

```
#define __MAP_FOLDR<n>(PREC_N, nil, one, cons, hd,
    tl, farg, f, arg, list) \
    cons(f(hd(list), arg),
        APP_EVAL_PARAM_MAP_FOLDR##PREC_N(PREC_N,
        PREC(PREC_N), nil, one, cons, hd, tl, farg,
         f, farg(arg), tl(list)))
#define APP_EVAL_PARAM_MAP_FOLDR<n>(PREC_N,
   PREC_PREC_N, nil, one, cons, hd, tl, farg, f,
   arg, list) \
    __MAP_FOLDR##PREC_N(PREC_PREC_N, nil, one,
        cons, hd, tl, farg, f, arg, list)
```

2.2 stack.h

2.2.1 Overview

Stack.h implements the STACK EL, using macros defined in macros.h. It is easy to translate a normal C function into a STACK EL coroutine. For example,

```
int add(int a, int b);
int main() {
   int c;
   c = add(1, 2);
   printf("1 + 2 = %d\n", x);
   return 0;
}
int add(int a, int b) {
   int c;
   c = a + b;
   return c;
}
```

can be translated into

```
void run(STACK_DEF) {

    #define LEVEL_add 1
    #define LEVEL_main 1

    PROTO(main, (), int)
    PROTO(add, (int a, int b), int)

        SEC_BEGIN(main, ())

    #define FUN main
        PARAMS()
        RETURNS(int)
    BEGIN
        DEFS(int, c)
        FUNC_CALL(add, (1, 2), VAR(c))
        printf("1 + 2 = %d\n", VAR(c));
    END
    #undef FUN

    #define FUN add
        PARAMS(int, a,
               int, b)
        RETURNS(int)
    BEGIN
        DEFS(int, c)
```

```
27          VAR(c) = VAR(a) + VAR(b);
28           RETURN(c);
29          END
30      #undef FUN
31
32          SEC_END
33  }
```

The enclosing C function "run" can be call the execute the STACK EL program. The STACK EL program is divided into two sections. The declaration section and the code section. The declaration section includes two types of declarations. The PROTO macro declares the prototype of a STACK EL function, similar to that of a C function. An exception is that that multiple return types can be declared, separated by commas, and the "void" return type is declared by not including a return type. The macro definition that looks like "#define LEVEL_..." declares the "level" of a STACK EL function. The basic idea is that the runtime stack is divided into different levels, the functions on the top of each "level" of the STACK EL functions can see each other's activation record, as if there are multiple separate stacks. For example,

```
1   #define LEVEL_f 1
2   #define LEVEL_g 1
3
4   PROTO(f, ())
5   PROTO(g, ())
6
7   SEC_BEGIN(main, ())
8
9   #define FUN f
10      PARAMS()
11      RETURNS()
12  BEGIN
13      DEFS(int, a)
14      VAR(a) = 1;
15      FUNC_CALL(g, ())
16  END
17  #undef FUN
18
19  #define FUN g
20  PARAMS()
21  RETURNS()
22  BEGIN
23      printf("a = %d\n", VAR_F1(f, a));
24  END
25  #undef FUN
26
27  SEC_END
```

The trade-offs are the more levels there are, the less efficient saving and restoring program states is. The number of levels is defined by a macro in stack.h

```
#define LEVEL_OF_NESTING 2
```

The code section starts with a call to the SEC_BEGIN macro, which specifies the main function and the initial arguments to the main function, and ends with a call to the SEC_END macro. The definition of a function starts with #define FUN followed by the name of the function, and ends with #undef FUN. The PARAMS macro defines parameters; the RETURNS macro defines return types; the DEFS macro defines local variables.

2.2.2 Saving and restoring program states

The SAVE_STATE macro and the RESTORE_STATE macro save and restore STACK EL program states, respectively. In the default setting, saved states are restored in a LIFO order. Because of the performance requirement, neither of these two macros actually copies any value of any STACK EL variables. Therefore, a variable should not be updated when a state involving that variable is saved. The next few subsections explains the details of this requirement.

2.2.3 How to use C variables

In STACK_EL code, C variables (local or global) = STACL EL global variables. They should be used with the same caveats as any global functions. In addition, they should only be used to pass values between two STACK EL function calls in a STACK EL function and they should not be used to pass values across function calls in a STACK EL function. The reason is that the SAVE_STATE and RESTORE_STATE macros do not save or restore C variables.

2.2.4 How to use STACK EL local variables

STACK EL local variables (We will omit "STACK EL") defined using PARAMS and DEFS should generally only be assigned once and should only be read after it is being assigned. Assigning a variable more than once may alter a previously saved state. For example, suppose a function looks like

```
VAR(c) = 1;
SAVE_STATE(rp)
rp:
// some code depending on VAR(c);
VAR(c) = 2;
```

When the save state is restored, the value of local variable "c" will be 2, which is inconsistent.

2.2.5 Level of nesting

The STACK EL supports multiple levels of nested functions. The semantics are as if each level has its own stack, and all stack tops are visible. The levels of nesting supported is given by LEVEL_OF_NESTING. To access var x at level n, VAR_N(x, n). To call a function f at any level n is called using the FUNC_CALL macro. All operators defaults to level 1. For example, VAR(x) = VAR_N(x, 1).

2.2.6 Calling a function and Tailcalling a function

A function is called by the FUNC_CALL macro. The first argument is the name of the callee, the second a tuple of the arguments, following which is a list of l-values that stores the return values of the function. Tail calls are made using the FUNC_TAIL_CALL macro.

2.2.7 Type Checking

The type checking feature of the stack EL checks function calls, tails call and function returns against the declared type of the involved functions at compile time.

2.2.8 Threads

By default, the saved states are restored in a LIFO order. The abstract data type used to store saved program states are called a threads. By default, there is only on thread, i.e., the current thread. STACK EL allows saving program states into a different thread. program states saved into a thread other than the current thread cannot be restored unless a switch thread action is performed. This is explained in my dissertation.

2.2.9 Global Mutable Data Dependency

Saved program state can be enabled/disabled depending on some global mutable data. This is a key feature that allows incremental computation when some global mutable data changes. This feature helps overcome the limitations of local variables. This is explained in my dissertation.

2.3 parsergen.h

Parsergen.h is a generic macro-based parser library. It support defining LL parsers in C code using a declarative syntax while supporting imperative semantic actions.

Each parser function corresponds to a non-terminal and is implemented as a C function. The PARSER_FUNC_PROTO macro declares a parser function that takes no arguments. There are multiple version of this macro. The PARSER_FUNC_PROTO<n> macro declares a parser function that

takes <n> arguments, current these macros are defined for $n \in \{1, 2\}$. As parser function begins with the PARSER_FUNC_BEGIN macro and ends with PARSER_FUNC_END macro. Similarly, there are multiple PARSER_FUNC_BEGIN<n> macros.

Within a parser function. The following macros can be used:

Macro	Usage
TTEXT(<token>)	Consume a text token, error if the next token does not match the argument.
TTEXT2(<token1>, <token2>)	Consume a text token, error if the next token does not match either argument.
NT(<nt>)	Parse non-terminal <nt> by calling the corresponding parser function
NT<n>(<nt>, ...)	Parse non-terminal <nt> by calling the corresponding parser function, with <n> arguments
TRY(<name>) <A1> OR(<name>) ... OR(<name>) <An> FINALLY(<name>) ... END_TRY(<name>)	Try parsing A1 | ... | An. <name> is the name of this try block. It should be unique within a parser function. The code following finally is executed no matter whether parsing is successful or not.
LOOP_BEGIN(<name>) ... DONE(<name>) ... LOOP_END(<name>)	Try parsing something like (...)*. <name> is the name of this loop. It should be unique withing a parser function. DONE signals the end of the loop. There can be multiple DONEs in a loop.
TTYPE(<type>)	Parse a token of a specific type. If successful. The token can be accessed from the "token" struct.
BUILD_NODE(<type>, <cons>, <loc>, <deg>, <consume>)	Bulid a node of type <type>, with value <cons>, and push it on to the node stack. <loc> is a Label object that stores the starting location of this node in the text being parsed. <deg> is the number of subnodes that are added to the newly-built node. These subnodes are taken from the top of the node stack. The topmost node will be a first subnode. <comsume> specifies the number of node that are popped from the node stack before the newly-built node is pushed on to the node stack

Macro	Usage
FPOS	Return a Label that represents the current code location. This object is stored in a shared data structure and it value can be changed by other macros.

To call a parser function use the following code:

```
Pointer e(<string or file>);
ParserContext pc(<region>);
parse<parser function name>(&e, &pc, ...);
```

2.4 Symbol Id

To improve efficiency, incOSHL as a unique 64-bit integral id to each symbol that it processes. An id encode several useful information that allows the prover to extract efficiently. The layout of the bits is shown in the following diagram:

```
bit     |0 ~ 7      |8 ~ 15     |16 ~ 22|23|24 ~ 31    |
func    |arity      |func index         |0 |
var     |var index                      |1 |version    |
```

For a function symbol or a predicate symbol, the first 8 bits store the arity of that function symbol, the next 15 bits store a unique index of that function symbol, and the next bits are all zero. For a variable symbol, the first 23 bits store a unique index of that variable symbol, the next bit stores one, and the next 7 bits store a version of this variable. Version is a feather that can be used to perform efficient clause instance search when there are non-ground literals in the model, and is not fully implemented in the current version of the prover. The layout of the bits for different types of symbols can be easily tweaked by change the macros in the def.h file.

2.5 Term Matrix

The term matrix is a huge hashtable that stores all generated terms. All terms should be generated from this matrix and not by calling the constructors of Term directly. The getFromTermMatrix function has three version:

getFromTermMatrix(char* key) returns a term from a key. The key is a byte array in the following format on a 64-bit system.

```
bit     |0 ~ 7       |8 ~ 15         |16 ~ 23        |...
type    |Symbol      |Term *         |Term *         |...
content |func symbol|ptr to 1st arg|ptr to snd arg|...
```

getFromTermMatrix(Symbol s) returns a term for a constant.

getFromTermMatrix(Symbol* f, Symbol * &next) returns a term from an array of symbols starting at "f". It returns the position where it stops in "next".

Term comparison can be done by direct pointer comparison.