Progess Report 1: Insight of SMACK

Presentor: Xie Li

April 13, 2021

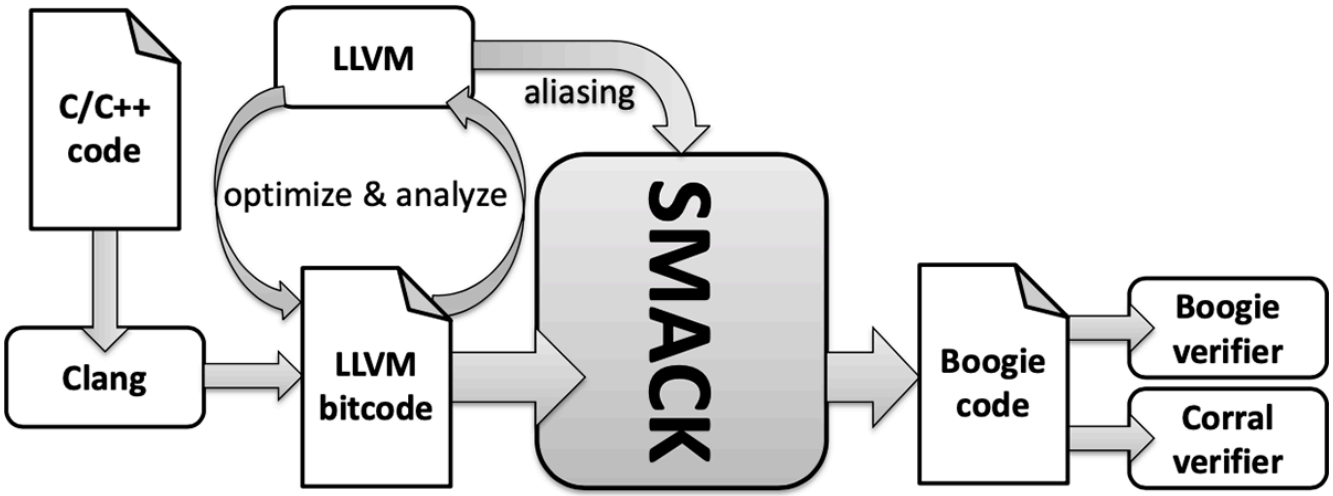
Overview

- Introduction to SMACK.
  - LLVM IR
  - Boogie IVL
  - Transformation.

Architecture

SMACK is an front-end tool capable of converting source code like C into IVL Boogie via LLVM IR.

Architecture of the toolchain:



- Frontend: Python
- Backend: A sequence of LLVM passes to for the transformation. Implemented in `llvm2bpl.c`.

Process of the Transformation

From LLVM bitcode to Boogie IVL:

1. `createInternalizePass()`, `createDeadCodeEliminationPass()` and `RemoveDeadDefs()`.
2. `createRemovePtrToIntPass()`, `createLowerSwitchPass()` and Passes for loop unrolling.
3. `NormalizeLoops()`, `SimplifyEV,IV`, `createCodifyStaticInitPass()`...
4. `MemorySafetyChecker()`, `IntegerOverflowChecker()`.
5. `SmackModuleGenerator()`, `BplFilePrinter()`.

Introduction to the SMACK: A Demo Example

C source code:

```c
#include "smack.h"
#include <stdlib.h>
#include "testlib.h"

typedef struct {
    int val;
    struct data_t* next;
} data_t;


int region_test(){
    int *m = 0;
    m = (int*)malloc(sizeof(int));
    free(m);
    return 0;
}

int dead_definition(){
    return 0;
}
```

```c
int main(){
    //char *what = (char*)malloc(sizeof(char));

    int *p;
    int x = 0;
    data_t data;
    data.val = 0;
    x = test_function_add(data.val,x);
    x = not_defined_function();
    p = (int*) malloc(sizeof(int));
    *p = 5;
    assert(x == 0 || x != 0);
    free(p);
    region_test();

}
```

LLVM IR

```
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 !dbg !44 {
  %1 = alloca %struct.data_t, align 8, !verifier.code !38
  call void @llvm.dbg.value(metadata i32 0, metadata !45, metadata !DIExpression()),
  !dbg !46, !verifier.code !38
  call void @llvm.dbg.declare(metadata %struct.data_t* %1, metadata !47, metadata
  !DIExpression()), !dbg !55, !verifier.code !38
  %2 = getelementptr inbounds %struct.data_t, %struct.data_t* %1, i32 0, i32 0, !dbg
  !56, !verifier.code !38
  store i32 0, i32* %2, align 8, !dbg !57, !verifier.code !38
  %3 = getelementptr inbounds %struct.data_t, %struct.data_t* %1, i32 0, i32 0, !dbg
  !58, !verifier.code !38
  %4 = load i32, i32* %3, align 8, !dbg !58, !verifier.code !38
  %5 = call i32 @test_function_add(i32 %4, i32 0), !dbg !59, !verifier.code !38
  call void @llvm.dbg.value(metadata i32 %5, metadata !45, metadata !DIExpression()),
  !dbg !46, !verifier.code !38
  %6 = call i32 (...) @not_defined_function(), !dbg !60, !verifier.code !38
  call void @llvm.dbg.value(metadata i32 %6, metadata !45, metadata !DIExpression()),
  !dbg !46, !verifier.code !38
  %7 = call noalias i8* @malloc(i64 4) #5, !dbg !61, !verifier.code !38
  %8 = bitcast i8* %7 to i32*, !dbg !62, !verifier.code !38
  call void @llvm.dbg.value(metadata i32* %8, metadata !63, metadata !DIExpression())
  !dbg !46, !verifier.code !38
  store i32 5, i32* %8, align 4, !dbg !64, !verifier.code !38
  br label %9, !dbg !65, !verifier.code !38

9:                                                ; preds = %0
  %10 = icmp eq i32 %6, 0, !dbg !66, !verifier.code !38
```

Boogie IVL

```
const main: ref;
axiom (main == $sub.ref(0, 5160));
procedure {:entrypoint} main()
  returns ($r: i32)
{
  var $p0: ref;var $p1: ref;var $p2: ref;var $i3: i32;var $i4: i32;var
  $i5: i32;
  var $p6: ref;var $p7: ref;var $i8: i1;var $i10: i1;var $i9: i1;var
  $i11: i32;var $i12: i32;var $p13: ref;var $i14: i32;
$bb0:
  call $initialize();
  assume {:sourceloc "../testcases/test1.c", 28, 10} true;
  assume {:verifier.code 0} true;
  call {:cexpr "smack:entry:main"} boogie_si_record_ref(main);
  assume {:verifier.code 0} true;
  call $p0 := $alloc($mul.ref(16, $zext.i32.i64(1)));
  assume true;
  assume {:sourceloc "../testcases/test1.c", 28, 10} true;
  assume {:verifier.code 0} true;
  $p1 := $add.ref($add.ref($p0, $mul.ref(0, 16)), $mul.ref(0, 1));
  assume {:sourceloc "../testcases/test1.c", 28, 14} true;
  assume {:verifier.code 0} true;
  $M.0 := $store.i32($M.0, $p1, 0);
  assume {:sourceloc "../testcases/test1.c", 29, 32} true;
  assume {:verifier.code 0} true;
```

LLVM API

LLVM API provides good support for developers on dealing with the optimizing and transforming of LLVM IR. In SMACK, before going through the passes, it first execute:

```
module = llvm::parseIRFile(InputFilename);
```

- Module
- Function
- BasicBlock
- Instruction

LLVM API

LLVM API provides various preset passes:

- `llvm::createGlobalDCEPass()`
- `llvm::createDeadCodeEliminationPass()`
- `llvm::createLowerSwitchPass()`
- `llvm::createLoopUnrollPass()`
- ...

If we wish to use these passes later, the functionalities of these passes need to be specified later.
LLVM API also provides convienient interface for extension:

- `llvm::InstVisitor<SmackInstGenerator>`
- `llvm::cl` for easy commandline argument manipulation.

A Program in Boogie AST

Definition (Boogie Program)

A program in boogie IVL is sequence of declarations.

On the implementation level: prelude + a list of `Decl` object.

- 7 Kinds of `Decl`: `ConstDecl`,`TypeDecl`,`AxiomDecl`, `ConstDecl`, `VarDecl`, `CodeDecl`, `FuncDecl`, `ProcDecl`
- `Stmt` are used in e.g. `FuncDecl`: `Assign`, `Assume`, `Assert`, `Goto`, `Code`, `Return`, `Call`, `Comment`, `Havoc`.
- `Expr`, `Attr`..

A Coarse Mapping from LLVM IR to Boogie AST

- Module
- Function, BasicBlock
- Instruction

- `Program`
- `FuncDecl, CodeDecl..`
- `Stmt`

Memory Model (No Reuse)

- Address are integers.
- One unbounded integer is stored at each address.
- Heap addresses are allocated in a strictly increasing fashion.
- Freed addresses are never reallocated.

Memory Model (No Reuse)

- Address space partition (Address A):

- $A > 0$
- $A = 0$
- $\texttt{GLOBALS\_BTM} \leq A < 0$
- $\texttt{GLOBALS\_BTM} - 32768 \leq A < \texttt{GLOBALS\_BTM}$
- $\texttt{EXTERNS\_BTM} \leq A < \texttt{GLOBALS\_BTM} - 32768$

- Heap
- `NULL`
- Static global storage
- Padding
- External global objects returned from external functions

A glimpse at the source here.

Transformation from LLVM IR to Boogie

`SmackModuleGenerator:` A pass generating a Boogie program from a module by replacing the module of llvm into a list of declarations.
`SmackRep:` the class responsible for the replacement.

- `globalDecl`: generate global declarations according to IR, add constraints that global declarations are put in the **global storage**.
- `ProcDecl`: insert a list of procedure declarations by replacing all functions in the module.
- `SmackInstGenerator`: generate `Stmt` for each procedure generated above by iterating all the instruction in functions.
- Generate prelude.

Generate Prelude

A `Prelude` is initialized by a `SmackRep` object. The generation include:

- `TypeDeclGen`: create the declaration for some basic types in boogie.
- `ConstDeclGen`: create constant declaration for integer, pointers...
- `MemDeclGen`: create memory regions.
- `IntOpGen`
- `PtrOpGen`
- `FpOpGen`

Have a look at the corresponding with the source code.

SmackRep

SmackRep(DataLayout*,Naming*,Program*,Regions*)

Regions

Region: Node + Offset + Length

- A bunch of instructions involving pointers are visited. (`Region::idx`).
- A region is created for the pointer operand (`Region::init`).
- Merge the created region into existing one if they overlap.

SMACK Header

```
#include "smack.h"
```

- `__SMACK_` + `code, mod, decl, value...`
- `__VERIFIER_` + `assume, assert, nondet`

Remaining Problems

- No systematic way to specify the passes.
- Details of instructions replacement need to be specified, especially those with
  - region operations,
  - assertion and assumption generation.
- Figure out the intention of passes.
- DSA, memory model..