| galois |

Lab: Introduction to SAW

The Software Analysis Workbench (SAW) is a tool for constructing mathematical models of the computational behavior of software, transforming these models, and proving properties about them. The models take the form of typed functional programs. Various external proof tools, including a variety of SAT and SMT solvers, can be used to prove properties about the functional models. SAW can construct models from arbitrary Cryptol programs, and from C and Java programs that have fixed-size inputs and outputs and that terminate after a fixed number of iterations of any loop (or a fixed number of recursive calls).

In many cases, such as C programs, SAW uses the Intermediate Representation (IR) llvm to provide the means to verify properties given a specification. The acronym llvm stands for low level virtual machine. Figure 1 below shows what the llvm project can do. The tool clang compiles C code to llvm IR. Then the llvm compiler can be used to continue compilation down to some architecture. As Figure 1 shows, there are tools for other languages that do the same thing. Thus, at the llvm level, the same function expressed in different languages, and even specifications, may be checked for equivalence.

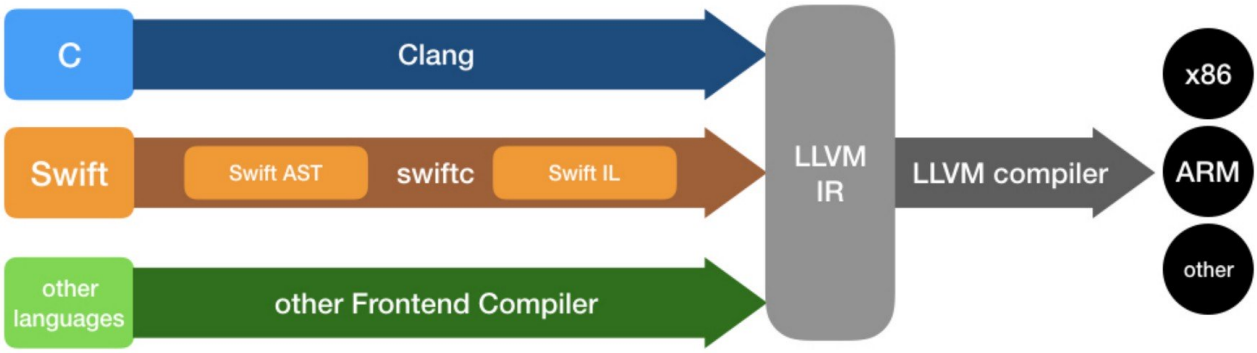

Figure 1: LLVM Frontend-Backend Compiler Architecture (from medium.com)

Consider the code of `add.c` below which just outputs the result of adding two 32 bit unsigned integers (the include files are not shown for purposes of simplification):

```
uint32_t add(uint32_t x, uint32_t y) {
    return x + y;
}

int main (int argc, char** argv) {
   uint32_t a = atol(argv[1]);
   uint32_t b = atol(argv[2]);
   printf("%u\n",add(a,b));
}
```

Clang is used as follows from the command line in Linux for producing `llvm` bitcode from compiled C code, the source of which is in file `add.c` (`clang` is available in Windows through Visual Studio – in case of difficulty in installing or using `clang`, `bc` files have been included in the `lab5A` directory. In Ubuntu Linux `clang` is installed using `sudo apt install clang-XX` where, in this lab, `XX` is 12):

```
clang-12 -g -O0 -c –emit-llvm add.c -o add.bc
```

For this to work, `clang-12` must be installed. Switch `-O0` means "no optimization": this level compiles the fastest and generates the most debuggable code. Switch `-01` instead means generated bitcode more closely matches the C/C++ source, making the results more comprehensible. Switch `-g` turns on debugging symbols so SAW can find source locations of functions, names of variables, etc.. In this example, compiled output is placed in `add.bc`. This is not human readable but can be converted to human readable by doing this: `llvm–dis add.bc` the result of which is `add.ll`. The extension `.bc` stands for bitcode. Consult the manual, Page 28, for helpul notes on compiling for SAW.

SAW can load a bitcode module with this:

```
m <- llvm_load_module "add.bc";
```

for the `clang-12` created `add.bc`.

SAW allows for specifying functions in `llvm`: the specification is defined in a `do` block. The `do` block is written in a `.saw` file along with a `llvm_load_module` line as above. For example:

```
let add_spec = do {
  ...
};
```

Inside the do block there are three sections: a specification of the initial state before execution of the function, a description of how to call the function within that state, and a specification of the expected final value of the program state. Part of the initial state specification is the declaration of `llvm` typed variables. For example,

```
x <- llvm_fresh_var "x" (llvm_int 32);
y <- llvm_fresh_var "y" (llvm_int 32);
```

which specifies two `llvm` variables of 32 bit integers. The `add_spec` function is to be called like this with `x` and `y` defined above:

```
llvm_execute_func [llvm_term x, llvm_term y];
```

The expected output for adding 32 bit integers `x` and `y` may be written like this:

```
llvm_return (llvm_term {{ x+y : [32]}});
```

The double brace block (`{{…}}`) encloses a cryptol expression. The `add_spec` specification may be checked for equivalence with the add function of `add.c`, which is in the `llvm` code of `add.bc`, using the following:

```
add_ov <- llvm_verify m "add" [] true add_spec z3;
```

where ‘`m`’ is the loaded llvm module from `add.bc`, ‘`add`’ is the function to be tested against the `add_spec`, ‘`[]`’ means skip any already-verified specifications that may be used for compositional verification (at this point a simple example is being considered and the use of

other verified specifications is not necessary), and ‘true’ means do path satisfiability checking (in this example false works just as well). Of course, ‘z3’ is the solver to be used for the equivalence check and add is being checked against ‘add_spec’ (cvc4 or abc works as well).

Exercise 1:
Create add.bc from add.c then put all of the above together in a file called add.saw, run saw, and observe the output. ■

Now consider the following slight modification to add.c, given In file add_ptr.c:

```
uint32_t add_in(uint32_t *x, uint32_t y) {
   *x += y;
   return *x;
}

int main (int argc, char** argv) {
   uint32_t a = atol(argv[1]);
   uint32_t b = atol(argv[2]);
   printf("%u\n",add_in(&a,b));
}
```

To deal with pointers use the following do block:

```
let ptr_to_fresh(name : String) (type : LLVMType) = do {
   x <- llvm_fresh_var name type;
   p <- llvm_alloc type;
   llvm_points_to p (llvm_term x);
   return (x, p);
};
```

This block returns an llvm variable x and an llvm pointer to x, namely p. It may be used in other do blocks like this:

```
(x,p) <- ptr_to_fresh "x" (llvm_int 32);
```

In this example the pointer is passed as argument so the following change may be made:

```
llvm_execute_func [p, llvm_term y];
```

Exercise 2:
Create add_ptr.bc from add_ptr.c then put all of the above together in a file called add_ptr.saw, run saw, and observe the output. ■

Now consider the following, given in file rotl.c:

```
uint32_t ROTL(uint32_t x, uint8_t r) {
   r = r % 32;
   uint32_t bottom = x >> (32 - r);
   uint32_t top = x << r;
   return bottom | top;
}

int main (int argc, char** argv) {
```

```
    uint32_t a = atol(argv[1]);
    uint32_t b = atol(argv[2]);
    printf("%u\n",ROTL(a,b));
}
```

Because `r` has a range from 0 to 31 due to this statement:

```
r = r % 32;
```

some precondition statements need to be added to the `llvm` specification. These look like this:

```
llvm_precond {{ 0 < r }};
llvm_precond {{ r < 32}};
```

Exercise 3:
Create `rotl.bc` from `rotl.c` then create `rotl.saw`, run `saw` on that file and observe the output. ■