| galois |

Lab: Functional Correctness

Software, hardware, and systems should do everything that they are intended to do and nothing more. The “nothing more” part of that statement is what is meant by safety and has been considered in the previous unit. Here the concern is proving that a specification is functionally equivalent to a function. Just a simple example to illustrate this.

Consider the following C code, in file `add.c`, that adds two 16 bit numbers:

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint16_t add(uint16_t x, uint16_t y) { return x + y; }

int main(int argc, char **argv) {
   if (argc != 3) {
      printf(“Usage: %s <number> <number>\n”,argv[0]);
      exit(0);
   }
   uint16_t a = (uint16_t)atoi(argv[1]);
   uint16_t b = (uint16_t)atoi(argv[2]);
   printf(“%d + %d = %d\n”,a,b,(uint16_t)add(a,b));
}
```

A corresponding Cryptol specification, in file `add.cry`, looks like this:

```
add : [16] -> [16] -> [16]
add x y = x+y
```

It is desired to show that the two are functionally equivalent even though possibly unacceptable results may occur due to integer overflow. In the previous unit showing code is safe took precedence. Now adherence to a specification takes precedence. The Cryptol ‘`add`’ is the specification.

Observe that, after compiling `add.c` and running `add 65000 65000` the result is 64464 instead of 130000 and loading `add.cry` in Cryptol and running `add 65000 65000` the result is also 64464. The concern here is equivalence and safety appears to take a back seat. But actually the specification should typically be safe unless intentionally not being such. Note, the `add` in Cryptol is safe because the signature says the output is 16 bits wide hence the *expected* overflow.

A SAW file that compares outputs of both functions begins with a familiar `add_spec` that creates two llvm fresh variables for `x` and `y`, invokes the C `add` on `x` and `y` and returns the Cryptol sum ‘`add x y`’ as a llvm term. This is used by `llvm_verify` to show equivalence between the C `add` and the Cryptol `add`. Of course, this must be preceeded by

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

Here is the SAW file `add.saw`:

```
import “add.cry”;

let add_spec = do {
   // Create fresh variables for x and y
   x <- llvm_fresh_var "x" (llvm_int 32);
   y <- llvm_fresh_var "y" (llvm_int 32);

   // Invoke the function with the fresh variables
   llvm_execute_func [llvm_term x, llvm_term y];

   // The function returns a value containing the sum of x and y
   llvm_return (llvm_term {{ add x y }});
};

let main : TopLevel () = do {
   m <- llvm_load_module “add.bc”;
   func_proof <- llvm_verify m “add” [] false add_setup yices;
   print “Done!”;
};
```

Run it like this:

```
saw add.saw
```

The result is this:

```
[16:49:48.263] Loading file ".../add.saw"
[16:49:48.363] Verifying add ...
[16:49:48.363] Simulating add ...
[16:49:48.365] Checking proof obligations add ...
[16:49:48.372] Proof succeeded! add
[16:49:48.372] Done!
```

The SAW/Cryptol suite does not have the same structure as most other common languages and some special things are added to deal with such differences. Consider how SAW handles global variables in C. The following, in file `global.c`, contains a global variable `g`:

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

uint32_t g;

void clear() {  g = 0; }
void set(uint32_t x) { g = x; }
uint32_t get() { return g; }

int main (int argc, char **argv) {
   uint32_t x;
   if (argc != 2) {
      printf("Usage: %s <number>\n",argv[0]);
      exit(0);
   }
   x = (uint32_t)atoi(argv[1]);
   clear();
   printf("clear(): %d\n",get());
   set(x);
   printf("set(x): %d\n",get());
}
```

Mutable global variables that are accessed in a function are allocated like this where `g` is the global variable in the C code above:

```
llvm_alloc_global "g";
```

This ensures that all global variables that might influence the function are accounted for explicitly in the specification and requires a corresponding `llvm_points_to` after `llvm_execute_func` which describes the new state of that global. The following does this for the `clear` function in the above code, so the new state of `g` is 0:

```
llvm_points_to (llvm_global "g") (llvm_term {{ 0 : [32] }});
```

Not using `llvm_points_to` potentially leads to unsoundness in the presence of compositional verification.

Thus, the SAW setup function for clear can be written as follows:

```
let clear_setup = do {
   llvm_alloc_global "g";
   llvm_execute_func [];  // clear takes no arguments
   llvm_points_to (llvm_global "g") (llvm_term {{ 0 : [32] }});
};
```

For the `set` function in the C code above a fresh `llvm` variable `x` is declared and used as argument in the `llvm_execute_func` line and as the state of global `g` in the `llvm_points_to` line. Otherwise, the setup function for `set` is the same as for `clear`:

```
let set_setup = do {
   llvm_alloc_global "g";
   x <- llvm_fresh_var "x" (llvm_int 32);
   llvm_execute_func [llvm_term x];
   llvm_points_to (llvm_global "g") (llvm_term x);
};
```

The next step is to prove the C functions `set` and `clear` match the above specifications. First, use clang to create `llvm` bitcode for `global.c` like this:

```
clang -g -O0 -emit -llvm -c global.c -o global.bc
```

Then SAW can extract `llvm` bitcode from named C functions from the result of this:

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

Verification of the `set` function looks like this:

```
llvm_verify m "set" [] false set_setup abc;
```

and for the `clear` function looks like this:

```
llvm_verify m "clear" [] false clear_setup abc;
```

Thus the `main : TopLevel()` looks like this:

```
let main : TopLevel () = do {
   m <- llvm_load_module "global.bc";
   llvm_verify m "clear" [] false clear_setup abc;
   llvm_verify m "set" [] false set_setup abc;
   print "Done.";
};
```

The `main` and `setup` sections above are in file `global.saw`. Running `saw global.saw` results in:

```
[14:06:56.823] Loading file ".../global.saw"
[14:06:56.882] Verifying clear ...
[14:06:56.882] Simulating clear ...
[14:06:56.884] Checking proof obligations clear ...
[14:06:56.885] Proof succeeded! clear
[14:06:56.885] Verifying set ...
[14:06:56.885] Simulating set ...
[14:06:56.886] Checking proof obligations set ...
[14:06:56.886] Proof succeeded! set
[14:06:56.886] Done.
```

Here is a more involved example, C code in file `global2.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

uint32_t x = 0;

uint32_t f ( uint32_t y ) {
   x = x + 1;
   return x + y ;
}

uint32_t g ( uint32_t z ) {
   x = x + 2;
   return x + z ;
}

uint32_t gf (uint32_t w) {
   return g(f(w));
}

int main (int argc, char **argv) {
   x=6;
   printf("f(1)=%d x=%d\n",f(1),x);
   x=6;
   printf("g(1)=%d x=%d\n",g(1),x);
   x=6;
   printf("gf(6)=%d x=%d\n", gf(6),x);
}
```

If globals were always initialized at the beginning of verification, specifications for both `f` and `g` would be provable. However, the results wouldn't truly be compositional. For instance, it's not the case that g(f(z)) == z + 3 for all z, because both `f` and `g` modify the global variable `x` in a way that crosses function boundaries. To deal with this the following SAW function is created:

```
let init_global name = do {
  llvm_alloc_global name;
  llvm_points_to ( llvm_global name )
                 ( llvm_global_initializer name );
};
```

`llvm_global_initializer` returns the value of the constant global initializer for the named global variable. Given this function, the specifications for `f` and `g` can make this reliance on the initial value of `x` explicit as in:

```
let f_setup = do {
   y <- llvm_fresh_var "y" (llvm_int 32);
   init_global "x";    // calls above function
   llvm_execute_func [ llvm_term y ];
   llvm_return (llvm_term {{ 1 + y:[32] }});
};
```

which initializes `x` to whatever it is initialized to in the C code at the beginning of verification. This specification is now safe for compositional verification: SAW won't use the specification `f_setup` unless it can determine that `x` still has its initial value at the point of a call to `f`. Here is the `main : TopLevel ()` that uses the above:

```
let main : TopLevel () = do {
   m <- llvm_load_module "global2.bc";
   prf1 <- llvm_verify m "f" [] true f_setup abc;
   print "Done!";
};
```

The reader can fill in the rest to cover the specification for `g` and for `g(f(w))`. For the composition g·f a Cryptol specification is created in file `gf.cry` with contents:

```
gf a = 4+2*x+a
  where
    x=6  // what x is initialized to in global2.c
```

Running `saw global2.saw` (with the g specification added) gives this:

```
[18:13:42.234] Loading file ".../global2.saw"
[18:13:42.335] Verifying f ...
[18:13:42.353] Simulating f ...
[18:13:42.357] Checking proof obligations f ...
[18:13:42.392] Proof succeeded! f
[18:13:42.439] Verifying g ...
[18:13:42.456] Simulating g ...
[18:13:42.459] Checking proof obligations g ...
[18:13:42.496] Proof succeeded! g
[18:13:42.544] Verifying gf ...
[18:13:42.560] Simulating gf ...
[18:13:42.564] Checking proof obligations gf ...
[18:13:42.599] Proof succeeded! gf
[18:13:42.599] Done!
```

Also instructive is to see how SAW handles C structs. Consider this C code in `struct.c`:

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct { uint32_t *x; } s;

uint32_t add_indirect(s *o) { return (o->x)[0] + (o->x)[1]; }
void set_indirect(s *o) { (o->x)[0] = 12;  (o->x)[1] = 6; }
s *s_id(s *o) { return o; }

int main (int argc, char **argv) {
   s p;
   uint32_t q[2];
   p.x = q;
   set_indirect(&p);
   printf("(%d,%d)\n", p.x[0], p.x[1]);
   printf("add_indirect=%d\n", add_indirect(&p)); }
```

Due to `add_direct` etc. `struct s` in this example is a 2 element array of 32 bit integers and has a `llvm` type

```
(llvm_array 2 (llvm_int 32));
```

In the SAW file, creating a fresh `llvm` variable of this type and a pointer to that variable is necessary and is accomplished with the `alloc_init` and `ptr_to_fresh` functions as follows.

Allocating space for a pointer is done with the following parameterized function `alloc_init`. Its parameters are a type `ty` and variable `v`. It outputs a pointer to space allocated to that variable.

```
let alloc_init ty v = do {
   p <- llvm_alloc ty;
   llvm_points_to p v;
   return p;
};
```

Function `alloc_init` is called by the following `ptr_to_fresh` function which returns a pair `(x,p)` where `x` is a fresh `llvm` variable and `p` is a pointer to space allocated for `x`.

```
let ptr_to_fresh n ty = do {
   x <- llvm_fresh_var n ty;
   p <- alloc_init ty (llvm_term x);
   return (x, p);
};
```

So, space may be allocated for an array consisting of 2 elements of 32 bit integers like this:

```
(x, px) <- ptr_to_fresh "x" (llvm_array 2 (llvm_int 32));
```

where `x` represents the array and `px` is a pointer to it. Now that array is placed in a `struct s` that consists only of that array which now has an `llvm` name `x` with pointer `px`. The type of `s` is

```
(llvm_struct_value [px]);
```

and the following allocates space for an object of type `struct s`:

```
po <- alloc_init (llvm_alias "struct.s") (llvm_struct_value [px]);
```

where `po` is a pointer to that object. Consider C function `set_indirect` which takes a pointer to a `struct s` object as argument. The `llvm_execute_func` then needs to be applied to `po` like this:

```
llvm_execute_func [po];
```

After execution of `set_indirect` there are two pointers to consider: the `po` pointer to the `s` object and the `px` pointer to the array within the `s` object. This requires

```
llvm_points_to po (llvm_struct_value [px]);
llvm_points_to px (llvm_term {{ [12, 6] : [2][32] }});
```

The `[12,6]` is just a sample initialization. The order of these two lines in `struct.saw` is not significant. The above six `llvm` lines will make up the setup function for the `set_indirect` function. The `add_indirect` and `s_id` functions have return values and their setup functions are slightly different. In the case of the setup for `add_indirect` the return value will be the same as adding `x@0` and `x@1` in Cryptol. The following in struct.saw expresses this:

```
llvm_return (llvm_term {{ x@0 + x@1 }});
```

For the setup function of `s_id` the return is a pointer to `s` object `o`. This is the following:

```
llvm_return po;  // recall po is established before llvm_execute_func
```

Create llvm bitcode with:

```
clang -g -O0 -emit -llvm -c struct.c -o struct.bc
```

Running `saw` on the complete `struct.saw` yields this output:

```
[12:47:10.462] Loading file ".../struct.saw"
[12:47:10.526] Verifying set_indirect ...
[12:47:10.526] Simulating set_indirect ...
[12:47:10.530] Checking proof obligations set_indirect ...
[12:47:10.530] Proof succeeded! set_indirect
[12:47:10.582] Verifying add_indirect ...
[12:47:10.583] Simulating add_indirect ...
[12:47:10.585] Checking proof obligations add_indirect ...
[12:47:10.639] Proof succeeded! add_indirect
[12:47:10.641] Verifying s_id ...
[12:47:10.641] Simulating s_id ...
[12:47:10.641] Checking proof obligations s_id ...
[12:47:10.641] Proof succeeded! s_id
```

A more complex use of `struct` is found in `dotprod_struct.c` where a dot product of two vectors up to the minimum dimension of both is computed. The `struct` holding vector information is

```
typedef struct {  uint32_t *elts;  uint32_t size;  } vec_t;
```

The contents of dotprod_struct.c is this:

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/param.h>

typedef struct {  uint32_t *elts;  uint32_t size;  } vec_t;

// Computes the dot product of the smaller vector and the
// equivalently-sized prefix of the larger vector.
uint32_t dotprod_struct(vec_t *x, vec_t *y) {
   uint32_t size = MIN(x->size, y->size);
   uint32_t res = 0;
   for(size_t i = 0; i < size; i++) res += x->elts[i] * y->elts[i];
   return res;
}

uint32_t dotprod_wrap(vec_t *x, vec_t *y) {  return dotprod_struct(x, y); }

int main(int argc, char **argv) {
   uint32_t a1[9] = { 1,2,3,4,5,6,7,8,9 };
   uint32_t a2[9] = { 2,4,6,8,0,8,6,4,2 };
   vec_t v1, v2;
   v1.elts = a1;
   v2.elts = a2;
   v1.size = 9;
   v2.size = 9;
   printf("%d\n", dotprod_wrap(&v1, &v2));
}
```

A specification for dot product is found in `dotprod.cry`.

```
zip : {n, a} (fin n) => (a -> a -> a) -> [n]a -> [n]a -> [n]a
zip f xs ys = [ f x y | x <- xs | y <- ys ]

sum : {n, a} (fin n, fin a) => [n][a] -> [a]
sum xs = ys!0
  where ys = [0] # [ x + y | x <- xs | y <- ys ]

dotprod : {n, a} (fin n, fin a) => [n][a] -> [n][a] -> [a]
dotprod xs ys = sum (zip (*) xs ys)
```

Function `zip` applies function `f` to pairs of elements from sequences `xs` and `ys` and outputs a sequence of those applications. For example, in Cryptol:

```
Main> zip (+) [1,2,3,4] [2,2,2,2]
[3, 4, 5, 6]
```

Function `sum` adds all elements of sequence `xs`. For example, in Cryptol:

```
Main> :s base=10
Main> sum [1,2,3,4,5:[32]]
15
```

Function `dotprod` applies `sum` to the sequence of products of elements of `xs` and `ys` obtained by `zip`. For example:

```
Main> dotprod [1,2,3,4,5,6,7,8,9:[32]] [2,4,6,8,0,8,6,4,2]
200
```

SAW file `dotprod_struct.saw` imports `dotprod.cry` and defines `alloc_init` and `ptr_to_fresh` as before. There is just one setup which starts like this:

```
let dotprod_setup n = do {
   let nt = llvm_term {{ `n : [32] }};  // A Cryptol function
```

The `n` is the length of the vectors to be operated on and automatically becomes a type variable in the corresponding Cryptol functions where the backtick and `:[32]` make the Cryptol `n` a bit vector of length 32. Then, as above, the following provide pointers to and space for the vectors that are operated on:

```
( xs , xsp ) <- ptr_to_fresh " xs " ( llvm_array n (llvm_int 32));
( ys , ysp ) <- ptr_to_fresh " ys " ( llvm_array n (llvm_int 32));
```

The following two lines create the structs that contain the input data. Recall the struct in the C code is defined as: `typedef struct { uint32_t *elts; uint32_t size; } vec_t;`

```
let xval = llvm_struct_value [ xsp, nt ];
let yval = llvm_struct_value [ ysp, nt ];
```

At this point space is allocated and pointers set to those allocations of the input data. But space and pointers need to be allocated for struct objects containing those data. The next two lines do this:

```
xp <- alloc_init (llvm_alias "struct.vec_t") xval;
yp <- alloc_init (llvm_alias "struct.vec_t") yval;
```

The `dotprod_wrap` function, which is the important one, will execute with parameters `xp` and `yp` like this:

```
llvm_execute_func [xp, yp];
```

The return value is taken from the Cryptol specification like this:

```
llvm_return (llvm_term {{ dotprod xs ys }});
```

Verification for the dot product of vectors of 100 elements then amounts to this:

```
let main : TopLevel () = do {
   m <- llvm_load_module "dotprod_struct.bc";
   llvm_verify m "dotprod_wrap" [] true (dotprod_setup 100) z3;
   print "Done!";
};
```

Run `saw dotprod_struct.saw` after running `clang` to get `dotprod_struct.bc`:

```
[15:14:45.486] Loading file ".../dotprod_struct.saw"
[15:14:45.655] Verifying dotprod_wrap ...
[15:14:45.675] Simulating dotprod_wrap ...
[15:14:45.730] Checking proof obligations dotprod_wrap ...
[15:14:45.789] Proof succeeded! dotprod_wrap
[15:14:45.789] Done!
```

Helpful SAW functions

```
// Given a value v of type ty, allocate memory for storing v and return a pointer
// to that memory
let alloc_init ty v = do {
   p <- crucible_alloc ty;
   crucible_points_to p v;
   return p;
};
```

```
// Given a value v of type ty, allocate memory for storing v and return a read-only
// pointer to that memory
let alloc_init_readonly ty v = do {
  p <- crucible_alloc_readonly ty;
  crucible_points_to p v;
  return p;
};
```

```
// Given a name n and a type ty, allocate a fresh variable x of type ty and return
// a tuple of x and a pointer to x.
let pointer_to_fresh n ty = do {
  x <- crucible_fresh_var n ty;
  p <- alloc_init ty (crucible_term x);
  return (x, p);
};
```

```
// Given a name n and a type ty, allocate a fresh variable x of type ty and return
// a tuple of x and a read-only pointer to x.
let ptr_to_fresh_readonly n ty = do {
  x <- crucible_fresh_var n ty;
  p <- alloc_init_readonly ty (crucible_term x);
  return (x, p);
};
```

```
// Given a name n and a value v, assert that the n has a value of v
let global_points_to n v = do {
   crucible_points_to (crucible_global n) (crucible_term v);
};
```

```
// Given a name n and a value v, declare that n is initialized, and assert that n
// has value v
let global_alloc_init n v = do {
   crucible_alloc_global n;
   global_points_to n v;
};
```