Getting Started with AutoCorres

Japheth Lim Rohan Jacob-Rao David Greenaway

October 9, 2015

Contents

1 Introduction

AutoCorres is a tool that attempts to simplify the formal verification of C programs in the Isabelle/HOL theorem prover. It allows C code to be automatically abstracted to produce a higher-level functional specification. AutoCorres relies on the C-Parser [3] developed by Michael Norrish at NICTA. This tool takes raw C code as input and produces a translation in SIMPL [4], an imperative language written by Norbert Schirmer on top of Isabelle. AutoCorres takes this SIMPL code to produce a "monadic" specification, which is intended to be simpler to reason about in Isabelle. The composition of these two tools (AutoCorres applied after the C-Parser) can then be used to reason about C programs.
This guide is written for users of Isabelle/HOL, with some knowledge of C, to get started proving properties of C programs. Using AutoCorres in conjunction with the verification condition generator (VCG) `wp`, one should be able to do this without an understanding of SIMPL nor of the monadic representation produced by AutoCorres. We will see how this is possible in the next chapter.

2 A First Proof with AutoCorres

We will now show how to use these tools to prove correctness of some very simple C functions.

2.1 Two simple functions: `min` and `max`

Consider the following two functions, defined in a file `minmax.c`, which (we expect) return the minimum and maximum respectively of two unsigned integers.

```
unsigned min(unsigned a, unsigned b) {
  if (a <= b) {
    return a;
  } else {
    return b;
  }
}

unsigned max(unsigned a, unsigned b) {
  return UINT_MAX - (
      min(UINT_MAX - a, UINT_MAX - b));
}
```

It is easy to see that `min` is correct, but perhaps less obvious why `max` is correct. AutoCorres will hopefully allow us to prove these claims without too much effort.

2.2 Invoking the C-parser

As mentioned earlier, AutoCorres does not handle C code directly. The first step is to apply the C-Parser[1] to obtain a SIMPL translation. We do this using the `install-C-file` command in Isabelle, as shown.

install-C-file *minmax.c*

For every function in the C source file, the C-Parser generates a corresponding Isabelle definition. These definitions are placed in an Isabelle "locale", whose name matches the input filename. For our file *minmax.c*, the C-Parser will place definitions in the locale *minmax*.[2]
For our purposes, we just have to remember to enter the appropriate locale before writing our proofs. This is done using the `context` keyword in Isabelle. Let's look at the C-Parser's outputs for `min` and `max`, which are contained in the theorems `min_body_def` and `max_body_def`. These are simply definitions of the generated names *min_body* and *max_body*. We can also see here how our work is wrapped within the *minmax* context.

context *minmax* **begin**

thm *min-body-def*

```
min-body ≡
TRY
  IF ´a ≤ ´b THEN
    creturn global-exn-var-'-update ret--unsigned-'-update a-'
  ELSE
    creturn global-exn-var-'-update ret--unsigned-'-update b-'
  FI;;
  Guard DontReach {} SKIP
CATCH SKIP
END
```

thm *max-body-def*

[1] http://ssrg.nicta.com.au/software/TS/c-parser

[2] The C-parser uses locales to avoid having to make certain assumptions about the behaviour of the linker, such as the concrete addresses of symbols in your program.

```
max-body ≡
TRY
  ´ret--unsigned :== CALL min-'proc(− 1 − ´a,− 1 − ´b);;
  creturn global-exn-var-'-update ret--unsigned-'-update
   (λs. − 1 − ret--unsigned-' s);;
  Guard DontReach {} SKIP
CATCH SKIP
END
```

end

The definitions above show us the SIMPL generated for each of the functions; we can see that C-parser has translated `min` and `max` very literally and no detail of the C language has been omitted. For example:

- C `return` statements have been translated into exceptions which are caught at the outside of the function's body;
- *Guard* statements are used to ensure that behaviour deemed 'undefined' by the C standard does not occur. In the above functions, we see that a guard statement is emitted that ensures that program execution does not hit the end of the function, ensuring that we always return a value (as is required by all non-`void` functions).
- Function parameters are modelled as local variables, which are setup prior to a function being called. Return variables are also modelled as local variables, which are then read by the caller.

While a literal translation of C helps to improve confidence that the translation is sound, it does tend to make formal reasoning an arduous task.

2.3 Invoking AutoCorres

Now let's use AutoCorres to simplify our functions. This is done using the `autocorres` command, in a similar manner to the `install_C_file` command:

autocorres *minmax.c*

AutoCorres produces a definition in the `minmax` locale for each function body produced by the C parser. For example, our `min` function is defined as follows:

context *minmax* **begin**
thm *min'-def*

$$min'\ a\ b \equiv if\ a \leq b\ then\ a\ else\ b$$

Each function's definition is named identically to its name in C, but with a prime mark (') appended. For example, our functions `min` above was named *min'*, while the function `foo_Bar` would be named *foo-Bar'*.
AutoCorres does not require you to trust its translation is sound, but also emits a *correspondence* or *refinement* proof, as follows:

Informally, this theorem states that, assuming the abstract function *min'* can be proven to not fail for a partciular input, then for the associated input, the concrete C SIMPL program also will not fault, will always terminate, and will have a corresponding end state to the generated abstract program. For more technical details, see [1] and [2].

2.4 Verifying `min`

In the abstracted version of *min'*, we can see that AutoCorres has simplified away the local variable reads and writes in the C-parser translation of `min`, simplified away the exception throwing and handling code, and also simplified away the unreachable guard statement at the end of the function. In fact, *min'* has been simplified to the point that it exactly matches Isabelle's built-in function *min*:

thm *min-def*

$$min\ a\ b = (if\ a \leq b\ then\ a\ else\ b)$$

So, verifying *min'* (and by extension, the C function `min`) should be easy:

lemma *min'-is-min*: *min' a b = min a b*
unfolding *min-def min'-def*
by (*rule refl*)

2.5 Verifying `max`

Now we also wish to verify that *max'* implements the built-in function *max*. *min'* was nearly too simple to bother verifying, but *max'* is a bit more complicated. Let's look at AutoCorres' output for `max`:

thm *max'-def*

$$max'\ a\ b \equiv -\ 1\ -\ min'\ (-\ 1\ -\ a)\ (-\ 1\ -\ b)$$

At this point, you might still doubt that *max'* is indeed correct, so perhaps a proof is in order. The basic idea is that subtracting from `UINT_MAX` flips the ordering of unsigned ints. We can then use *min'* on the flipped numbers to compute the maximum.
The next lemma proves that subtracting from `UINT_MAX` flips the ordering. To prove it, we convert all words to *int*'s, which does not change the meaning of the statement.

```
lemma n1-minus-flips-ord:
  ((a :: word32) ≤ b) = ((-1 - a) ≥ (-1 - b))
  apply (subst word-le-def)+
  apply (subst word-n1-ge [simplified uint-minus-simple-alt])+
```

Now that our statement uses *int*, we can apply Isabelle's built-in `arith` method.

```
  apply arith
  done
```

And now for the main proof:

```
lemma max'-is-max: max' a b = max a b
  unfolding max'-def min'-def max-def
  using n1-minus-flips-ord
  by force

end
```

In the next section, we will see how to use AutoCorres to simplify larger, more realistic C programs.

3 More Complex Functions with AutoCorres

In the previous section we saw how to use the C-Parser and AutoCorres to prove properties about some very simple C programs.
Real life C programs tend to be far more complex however; they read and write to local variables, have loops and use pointers to access the heap. In this section we will look at some simple programs which use loops and access the heap to show how AutoCorres can allow such constructs to be reasoned about.

3.1 A simple loop: `mult_by_add`

Our C function `mult_by_add` implements a multiply operation by successive additions:

```
unsigned mult_by_add(unsigned a, unsigned b)
{
    unsigned result = 0;
    while (a > 0) {
        result += b;
        a--;
    }
    return result;
}
```

We start by translating the program using the C parser and AutoCorres, and entering the generated locale `mult_by_add`.

install-C-file *mult-by-add.c*
autocorres [*ts-rules* = *nondet*] *mult-by-add.c*

The C parser produces the SIMPL output as follows:

thm *mult-by-add-body-def*

```
mult-by-add-body ≡
TRY
  ´result :== scast 0;;
  WHILE scast 0 < ´a DO
    ´result :== ´result + ´b;;
    ´a :== ´a − scast 1
  OD;;
  creturn global-exn-var-'-update ret--unsigned-'-update result-';;
  Guard DontReach {} SKIP
CATCH SKIP
END
```

Which is abstracted by AutoCorres to the following:

thm *mult-by-add'-def*

```
mult-by-add' a b ≡
do (a, result) ←
    whileLoop (λ(a, result) b. 0 < a)
     (λ(a, result). return (a − 1, result + b))
    (a, 0);
    return result
od
```

In this case AutoCorres has abstracted `mult_by_add` into a *monadic* form. Monads are a pattern frequently used in functional programming to represent imperative-style control-flow; an in-depth introduction to monads can be found elsewhere.

The monads used by AutoCorres in this example is a *non-deterministic state monad*; program state is implicitly passed between each statement, and results of computations may produce more than one (or possibly zero) results[3].
(* FIXME : probably describe below in further detail. *) The bulk of *mult-by-add'* is wrapped inside the *whileLoop monad combinator*, which is really just a fancy way of describing the method used by AutoCorres to encode (potentially non-terminating) loops using monads.
If we want to describe the behaviour of this program, we can use Hoare logic as follows:

$$\{\!|P|\!\}\ \textit{mult-by-add'}\ a\ b\ \{\!|Q|\!\}$$

This predicate states that, assuming P holds on the initial program state, if we execute *mult-by-add'* a b, then Q will hold on the final state of the program.
There are a few details: while P has type $'s \Rightarrow \textit{bool}$ (i.e., takes a state and returns true if it satisifies the precondition), Q has an additional parameter $'r \Rightarrow 's \Rightarrow \textit{bool}$. The additional parameter $'r$ is the *return value* of the function; so, in our `mult_by_add'` example, it will be the result of the multiplication. For example one useful property to prove on our program would be:

$$\{\!|\lambda s.\ \textit{True}|\!\}\ \textit{mult-by-add'}\ a\ b\ \{\!|\lambda r\ s.\ r = a * b|\!\}$$

That is, for all possible input states, our `mult_by_add'` function returns the product of a and b.
Unfortunately, this is not sufficient. As mentioned in the previous section, AutoCorres produces a theorem for each function it abstracts stating that, *assuming the function does not fail*, then the generated function is a valid abstraction of the concrete function. Thus, if we wish to reason about our concrete C function, we must also show non-failure on the abstract program. This can be done using the predicate *no-fail* as follows:

$$\bigwedge a\ b.\ \textit{no-fail}\ (\lambda s.\ \textit{True})\ (\textit{mult-by-add'}\ a\ b)$$

Here $\lambda s.\ \textit{True}$ is the precondition on the input state.
Instead of proving our Hoare triple and *no-fail* separately, we can prove them together using the "valid no fail" framework as follows:

$$\{\!|P|\!\}\ f\ \{\!|Q|\!\}! \equiv \{\!|P|\!\}\ f\ \{\!|Q|\!\} \wedge \textit{no-fail}\ P\ f$$

[3]Non-determinism becomes useful when modelling hardware, for example, where the exact results of the hardware cannot be determined ahead of time.

Our proof of *mult-by-add′* could then proceed as follows:

lemma *mult-by-add-correct*:
$\{\!|\ \lambda s.\ True\ |\!\}\ mult\text{-}by\text{-}add'\ a\ b\ \{\!|\ \lambda r\ s.\ r = a * b\ |\!\}!$

Unfold abstracted definition

apply (*unfold mult-by-add′-def*)

Annotate the loop with an invariant and a measure.

apply (*subst whileLoop-add-inv*
[**where** $I{=}\lambda(a', result)\ s.\ result = (a - a') * b$
and $M{=}\lambda((a', result), s).\ a'$])

Run the "weakest precondition" tool `wp`.

apply *wp*

Solve the program correctness goals.

apply (*simp add*: *field-simps*)
apply *unat-arith*
apply (*auto simp*: *field-simps not-less*)
done

The proof is straight-forward, but uses a few different techniques: The first is that we annotate the loop with a loop invariant and a measure, using the rule $whileLoop\ B\quad C = (\lambda x.\ whileLoop\text{-}inv\ B\ C\ x\ I\ (measure'\ M))$. We then run the `wp` tool which applys a large set of *weakest precondition* rules on the program[4]. We finially discharge the remaining subgoals left from the `wp` tool using `auto`, and our proof is complete.
In the next section, we will look at how we can use AutoCorres to verify a C program that reads and writes to the heap.

3.2 `swap`

Here, we use AutoCorres to verify a C program that reads and writes to the heap. Our C function, `swap`, swaps two words in memory:

```
void swap(unsigned *a, unsigned *b)
{
    unsigned t = *a;
    *a = *b;
    *b = t;
}
```

[4]This set of rules includes a rule which can handle annotated *whileLoop* terms, but will not attempt to process *whileLoop* terms without annotations.

Again, we translate the program using the C parser and AutoCorres.

install-C-file *swap.c*
autocorres [*heap-abs-syntax*, *ts-rules* = *nondet*] *swap.c*

Most heap operations in C programs consist of accessing a pointer. AutoCorres abstracts the global C heap by creating one heap for each type. (In our simple `swap` example, it creates only a *word32* heap.) This makes verification easier as long as the program does not access the same memory as two different types.
There are other operations that are relevant to program verification, such as changing the type that occupies a given region of memory. AutoCorres will not abstract any functions that use these operations, so verifying them will be more complicated (but still possible).

The C parser expresses `swap` like this:

thm *swap-body-def*

```
swap-body ≡
TRY
  Guard C-Guard {|c-guard ´a|} (´t :== h-val (hrs-mem ´t-hrs) ´a);;
  Guard C-Guard {|c-guard ´a|}
   (Guard C-Guard {|c-guard ´b|}
     (´globals :==
        t-hrs-'-update
         (hrs-mem-update (heap-update ´a (h-val (hrs-mem ´t-hrs) ´b)))));;
  Guard C-Guard {|c-guard ´b|}
   (´globals :== t-hrs-'-update (hrs-mem-update (heap-update ´b ´t)))
CATCH SKIP
END
```

AutoCorres abstracts the function to this:

thm *swap'-def*

```
swap' a b ≡
do guard (λs. is-valid-w32 s a);
   t ← gets (λs. s[a]);
   guard (λs. is-valid-w32 s b);
   modify (λs. s[a := s[b]]);
   modify (λs. s[b := t])
od
```

There are some things to note:
The function contains guards (assertions) that the pointers `a` and `b` are valid. We need to prove that these guards never fail when verifying `swap`.

Conversely, when verifying any function that calls `swap`, we need to show that the arguments are valid pointers.
We saw a monadic program in the previous section, but here the monad is actually being used to carry the program heap.
(Something about heap syntax here.)

Now we prove that `swap` is correct. We use x and y to "remember" the initial values so that we can talk about them in the postcondition.

lemma $\{\!|\ \lambda s.\ \textit{is-valid-w32}\ s\ a \wedge s[a] = x \wedge \textit{is-valid-w32}\ s\ b \wedge s[b] = y\ |\!\}$
  $\textit{swap}'\ a\ b$
 $\{\!|\ \lambda\text{-}\ s.\ s[a] = y \wedge s[b] = x\ |\!\}!$
apply (*unfold swap′-def*)
apply *wp*
apply *clarsimp*

The C parser and AutoCorres both model the C heap using functions, which takes a pointer to some object in memory. Heap updates are modelled using the functional update *fun-upd*:

> $f(a := b) \equiv \lambda x.\ \textit{if}\ x = a\ \textit{then}\ b\ \textit{else}\ f\ x$

To reason about functional updates, we use the rule fun_upd_apply.

apply (*simp add*: *fun-upd-apply*)
done

Note that we have "only" proved that the function swaps its arguments. We have not proved that it does *not* change any other state. This is a typical *frame problem* with pointer reasoning. We can prove a more complete specification of `swap`:

lemma $(\bigwedge x\ y\ s.\ P\ (s[a := x][b := y]) = P\ s) \Longrightarrow$
 $\{\!|\ \lambda s.\ \textit{is-valid-w32}\ s\ a \wedge s[a] = x \wedge \textit{is-valid-w32}\ s\ b \wedge s[b] = y \wedge P\ s\ |\!\}$
  $\textit{swap}'\ a\ b$
 $\{\!|\ \lambda\text{-}\ s.\ s[a] = y \wedge s[b] = x \wedge P\ s\ |\!\}!$
apply (*unfold swap′-def*)
apply *wp*
apply (*clarsimp simp*: *fun-upd-apply*)
done

In other words, if predicate P does not depend on the inputs to `swap`, it will continue to hold.
Separation logic provides a more structured approach to this problem.

References

[1] David Greenaway, June Andronick, and Gerwin Klein. Bridging the gap: Automatic verified abstraction of C. In Lennart Beringer and Amy Felty, editors, *Proceedings of the 3rd International Conference on Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 99–115, Princeton, New Jersey, August 2012. Springer.

[2] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 429–439, Edinburgh, UK, June 2014. ACM. doi:10.1145/2594291.2594296.

[3] Michael Norrish. C-to-Isabelle parser, version 1.13.0, May 2013. Accessed July 2014. URL: http://ertos.nicta.com.au/software/c-parser/.

[4] Norbert Schirmer. A sequential imperative programming language syntax, semantics, hoare logics and verification environment. *Archive of Formal Proofs*, February 2008. Formal proof development. URL: http://afp.sf.net/entries/Simpl.shtml.