HENSOLDT

Detect and Protect.

Hands on: Watching software being verified

Dr. Jaap Boender

This presentation

- Presentation of example
  - C code
  - State
  - Abstract specification (monadic, in Isabelle)
  - Monadic version of the C program
  - Refinement proof

The C program

```c
unsigned int test[5] = {1, 2, 3, 4, 5};

void add_array (unsigned int l, unsigned int n)
{
  unsigned int i;
  for (i = 0; i < l; i++)
    test[i] += n;
}
```

- Program adds *n* to each element of an array (of length *l*)
- The state of this program is the contents of the array *test*

The C program

SIMPL version

```
add_array_body ≡ TRY
                   lvar_nondet_init i_' i_'_update;;
                   ´i :== SCAST(32 signed → 32) 0;;
                   WHILE ´i < ´l DO
                     Guard ArrayBounds ⦃´i < SCAST(32 signed → 32) 5⦄ (Guard ArrayBounds ⦃´i < SCAST(32 signed → 32) 5⦄ (´globals :== test_'_update (λ_. Arrays.update ´test (unat ´i) (´test.[unat ´i] + ´n))));;
                     ´i :== ´i + SCAST(32 signed → 32) 1
                   OD
                 CATCH SKIP
                 END
```

- SIMPL translates C into an Isabelle representation
- 'Untrusted', so the translation needs to be as simple as possible
- Shallow embedding of datatypes
- Not necessarily easy to work with

The C program

AutoCorres version

```
add_array' ?l ?n ≡ do i <- whileLoop (λi s. i < ?l)
                              (λi. do guard (λs. i < 5);
                                      modify (λs. s(|test_'' := Arrays.update (test_'' s) (unat i) (test_'' s.[unat i] + ?n)|));
                                      return (i + 1)
                                   od)
                             0;
                        skip
                     od
```

- Generated by AutoCorres from the SIMPL version
- Uses monads: canonical way of dealing with state
- Proof of equivalence between SIMPL and AutoCorres versions automatically generated

The Isabelle specification

The state

```
record abstract_state =
  test :: "nat list"
```

- Abstract state: Isabelle record
- Native Isabelle list rather than a C array
  - (we'll need to keep an eye on length!)
- Use nat rather than word

The Isabelle specification

The monadic implementation

```
definition add_list :: "nat ⇒ (abstract_state, unit) nondet_monad" where
  "add_list n ≡ modify (update_test (map (λx. x + n)))"
```

- Uses monads as well
  - *modify* applies a function to the state

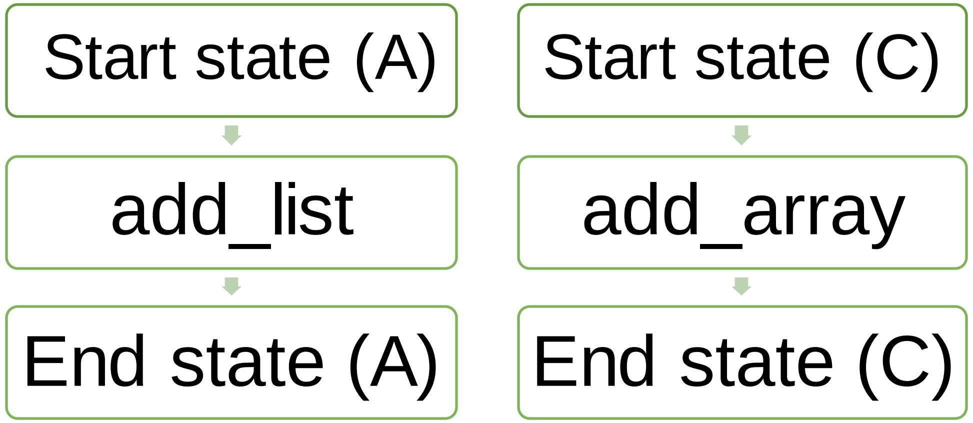
The Isabelle specification

The state relation

```
definition state_relation where
  "state_relation ≡ {(sₐ, s꜀). length (test sₐ) = length_of_array test_'' ∧
    (∀x<length_of_array test_''. test sₐ ! x = unat (test_'' s꜀.[x]))}"
```

- The state relation links the abstract state with the C state
  - In short: the *test* list has the same length and contents as the *test* array

Refinement proof



- A refinement proof shows that two functions are semantically equivalent
- Given two related start states, the states after execution of both abstract and C programs will be related

Refinement proof

Isabelle formulation

```
lemma add_refine:
  "⟦ l = of_nat (length_of_array (test_'')); n = unat n' ⟧ ⟹
  corres_underlying
    state_relation
    True True
    dc
    ⊤ (λs. ∀x<length_of_array test_''. unat (test_'' s.[x]) + unat n' < 2^LENGTH(32))
    (add_list n) (add_array' l n')"
```

- Some assumptions about the parameters (length of list, *nat* vs. *word*)
- Don't care about the return value (*dc*)
- One precondition for the C state: no overflow

Refinement proof

Goal to solve

```
goal (1 subgoal):
 1. ⋀sₐ s꜀.
       ⟦l = 5; n = unat n'; (sₐ, s꜀) ∈ state_relation; ∀x<5. unat (test_'' s꜀.[x]) + unat n' < 4294967296⟧
       ⟹ corres_underlying state_relation True True dc (λs. s = sₐ) (λs. s = s꜀ ∧ (∀x<5. unat (test_'' s.[x]) + unat n' < 4294967296)) (modify (update_test (map (λx. x + unat n'))))
           (whileLoop (λr s. r < 5)
             (λr. do y <- modify (λs. s⦇test_'' := Arrays.update (test_'' s) (unat r) (test_'' s.[unat r] + n')⦈);
                     return (r + 1)
                  od)
            0)
```

- Some administration beforehand: keep reference to initial state and precondition
- Now to prove: *modify* does the same thing as our loop
- For this, we use an **invariant**

Refinement proof

Loop invariant

- We do not necessarily know how many iterations a loop will go through (can depend on the state!)
- We must provide an invariant (for the loop postcondition) and a measure (for termination)
  - Invariant must hold **before** the loop
  - Invariant must hold **after each iteration** of the loop (assuming that it held before the iteration)
  - Invariant must hold **after** the loop
  - Measure must be a **well-founded relation** (for example the strict order on natural numbers)

```
goal (1 subgoal):
 1. ⋀s_a s_c.
       ⟦l = 5; n = unat n'; (s_a, s_c) ∈ state_relation; ∀x<5. unat (test_'' s_c.[x]) + unat n' < 4294967296⟧
       ⟹ corres_underlying state_relation True True dc (λs. s = s_a) (λs. s = s_c ∧ (∀x<5. unat (test_'' s.[x]) + unat n' < 4294967296)) (modify (update_test (map (λx. x + unat n'))))
           (whileLoop_inv (λr s. r < 5) (λr. do y <- modify (λs. s⦇test_'' := Arrays.update (test_'' s) (unat r) (test_'' s.[unat r] + n')⦈);
                                                return (r + 1)
                                             od)
             0 (λi s. i ≤ 5 ∧ (∀x<unat i. test_'' s.[x] = test_'' s_c.[x] + n') ∧ (∀x<5. unat i ≤ x ⟶ test_'' s.[x] = test_'' s_c.[x])) (measure' (λ(i, s). 5 - unat i)))
```

Refinement proof

Sketch of rest of proof

- Loop invariant holds at beginning (i = 0, so trivially true)
- Loop invariant holds throughout iterations
  - All x < i will not have changed, so invariant holds
  - If x = i, then we have just modified the x'th element, so we can use that knowledge
- After the loop, x = 5, so we know that all 5 elements have been updated
- State relation therefore holds

Refinement proof

What do we now know?

- There is a refinement between *add_list* and *add_array*
- This means that any properties that hold for *add_list* also hold for *add_array*
- This allows a separation of concerns strategy:
  - Proof of desired properties can be done using *add_list* (much easier)
  - All the finicky C semantics are dealt with in the refinement proof

Conclusion

- In order to verify a program using this technique:
  - Write an abstract version in Isabelle (using monads and Isabelle's native data structures)
  - Define a relation between the abstract state and the C state
  - Show refinement between the abstract functions and the C functions
  - (Not shown) Fit all functions together into a state machine
- Acta est fabula, plaudite

Dr. Jaap Boender
Formal Verification Engineer
jacob.boender@hensoldt-cyber.de

HENSOLDT Cyber GmbH
Willy-Messerschmitt-Straße 3
82024 Taufkirchen
hensoldt-cyber.com

This document and its content is the property of HENSOLDT Cyber GmbH. It shall not be communicated to any third party without the owner's written consent. © Copyright HENSOLDT Cyber GmbH 2020. All rights reserved.