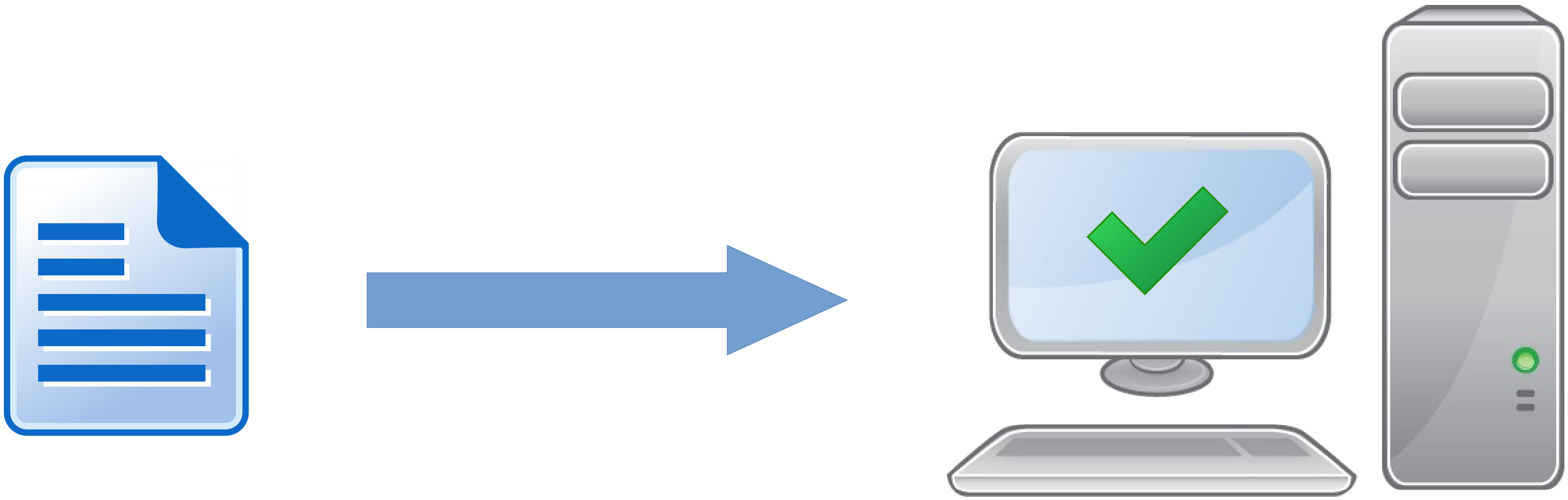
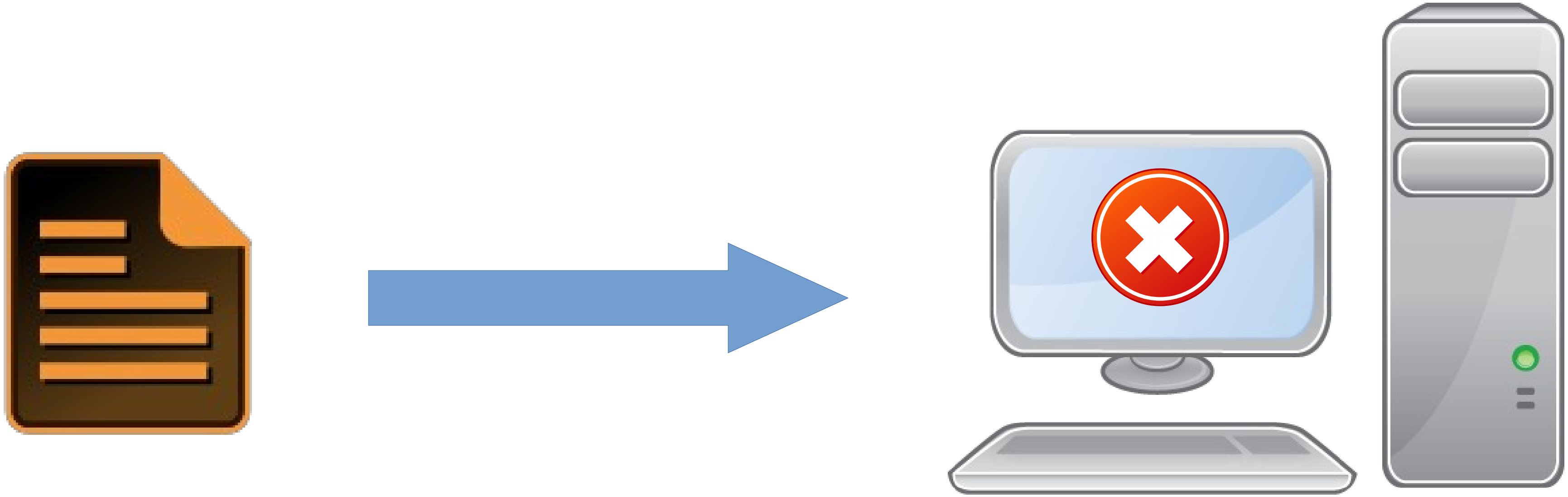
Verifying C Data-types for Cogent

Louis Francis Cheung (z5062193)

How Programs Should Work



What Normally Happens

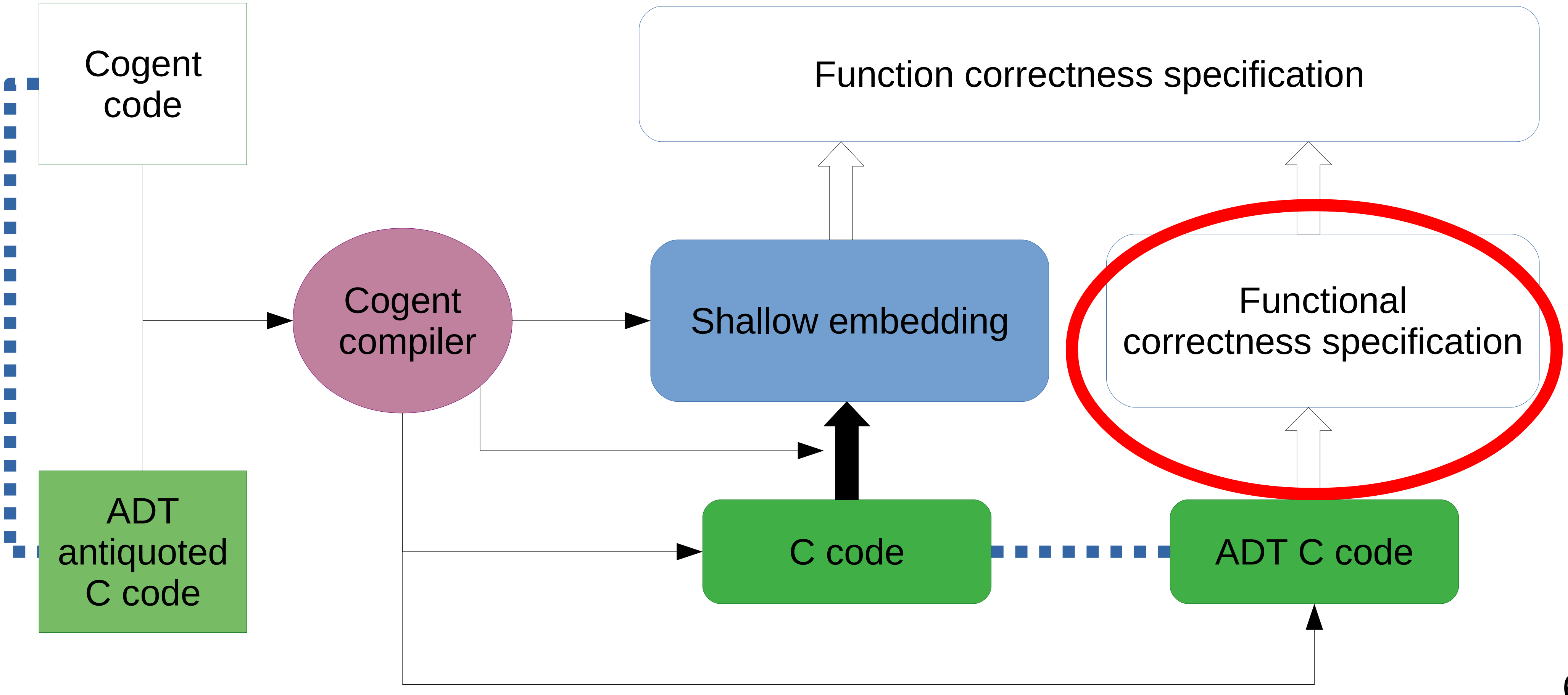


Verification = Trust

Cogent

- Restricted, pure, polymorphic, functional language with linear types and no runtime environment nor garbage collector
- Has a certifying compiler
  - Isabelle/HOL shallow embedding
  - C code
  - Refinement proof
- No native support for recursion
- Use FFI to call C functions that implement ADTs and iterators
  - The C functions need to be verified manually

The Big Picture



Expected Outcomes I

- Define a functional correctness specification for 32-bit word arrays
- Verify functional correctness of the 32-bit word array implementation
- Verify that the 32-bit word array implementation satisfies Cogent’s frame constraints

Expected Outcomes II

- Formulate a generic specification for word arrays
- Formulate generic proof techniques and requirements to prove functional correctness of word arrays
- Formulate generic proof techniques and requirements to prove frame constraint satisfiability of word arrays

What Has Been Achieved?

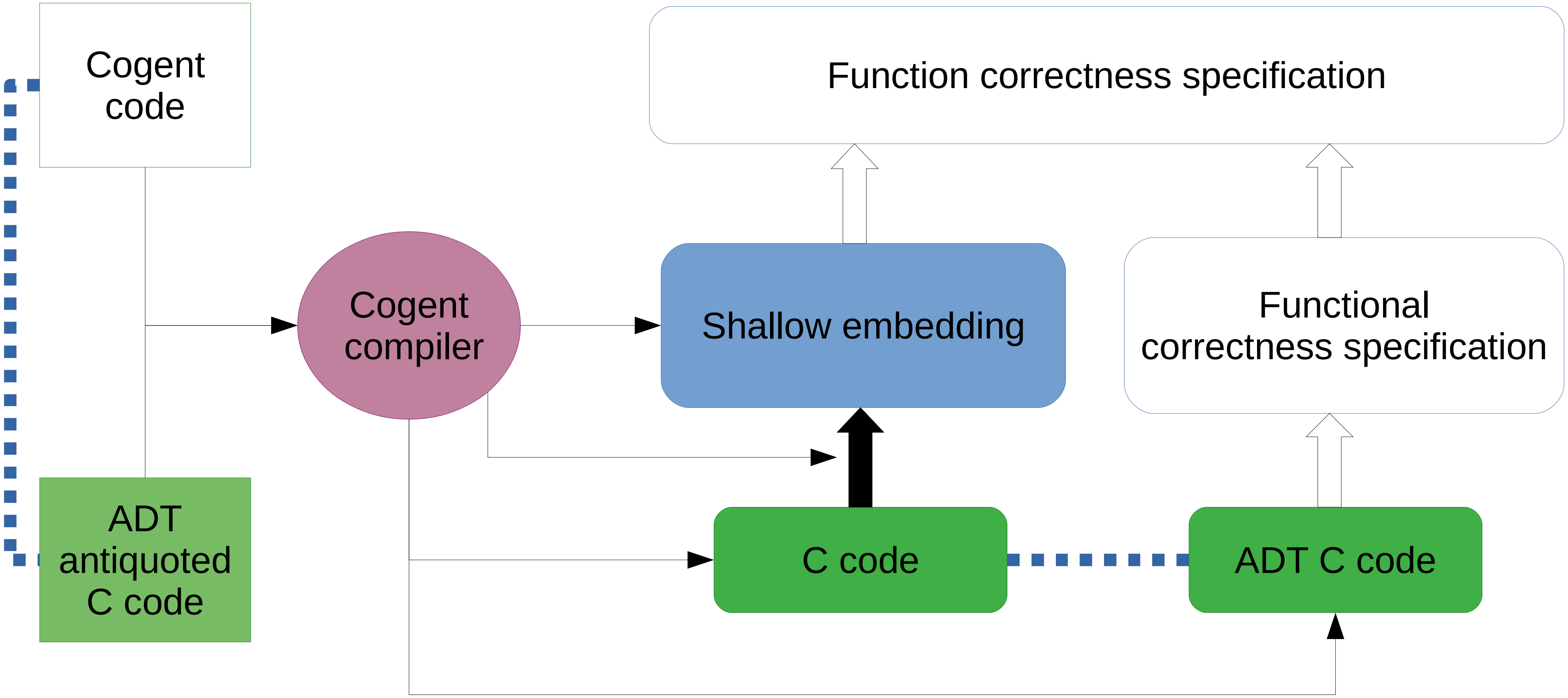
- Defined a functional correctness specification for 32-bit word arrays
- Verified functional correctness for the get, length, put and fold
- Verified frame correctness for get, length, put and fold
- 32-bit word array proofs also work with slight modifications for other standard word lengths

Methodology

Functional Correctness Specification

- 2 Choices:
  1) Define an abstraction from word arrays to an abstract data type (Isabelle/HOL lists)
  2) Define high level HOL properties for word arrays

Why Option 1?



Cogent code
Function correctness specification
Cogent compiler
Shallow embedding
Functional correctness specification
ADT antiquoted C code
C code
ADT C code

Frame Constraints for C Pointers

- Inertia
  - $p \notin w_i \cup w_o \Rightarrow \mu_i(p) = \mu_o(p)$
- Leak freedom
  - $p \in w_i \wedge p \notin w_o \Rightarrow \mu_o(p) = \perp$
- Fresh allocation
  - $p \notin w_i \wedge p \in w_o \Rightarrow \mu_i(p) = \perp$

$\text{Pointer} : p$

$\text{Input writable pointer set} : w_i$

$\text{Output writable pointer set} : w_o$

$\text{Input heap function} : \mu_i$

$\text{Output heap function} : \mu_o$

Generate the C files

```
$ty:(WordArray a) $id:wordarray_put2($ty:(WordArrayPutP a) args)
{
        if (likely(args.idx < (args.arr)->len)) {
                args.arr->values[args.idx] = args.val;
        }

        return args.arr;
}
```

```
WordArray_u32 *wordarray_put2_0(t2 args)
{
    if (__builtin_expect(!!(args.idx < args.arr->len), 1))
        args.arr->values[args.idx] = args.val;
    return args.arr;
}
```

Results

Bug Discovery

```
struct WordArray_u32 {
    int len;
    u32 *values;
} ;
```

Points of Interest

Pointer Arithmetic

- Accessing the i[th] element in an array
  - $p+k\times i$
- Issues
  - $a\neq b\Rightarrow p+k\times a\neq p+k\times b$?

Word Arithmetic

- $k \times a \equiv k \times b \quad mod\, n \quad \Rightarrow \quad a \equiv b \quad mod\, n$?
- Only if *k* is coprime with *n*
  - $2 \times 2 \equiv 2 \times 6 \quad mod\, 8$

What’s the Issue

- 32-bit words are 4 bytes in size
- Word arithmetic in a 32-bit architecture modulo 2^{32}
- Overflow can allow 2 distinct indices to have the same addresses in memory
- Assume that 4 times the length of a 32-bit word array is less than 2^{32}

Frame Constraints

- Different memory models
  - Cogent has a single heap
  - Autocorres abstracts the heap as different typed heaps
- Which pointers are writeable?
- Should a Cogent pointer equate to single C pointer or to a set of C pointers?

Conclusions and Future Work

Specification

- Functional correctness specification for 32-bit word arrays can be reused for arbitrary word length arrays (with a few tweaks)

Proof Techniques

- Most of the proofs can be reused for arbitrary word length arrays (with a few tweaks)

Frame Constraint Satisfiability

- Develop a tactic to automatically extract the writeable pointer sets and to define the frame constraints for each type