Quarterly LangSec Review #1

Dancing, Type Systems, and Quantum Computers

Charles Averill

Dallas Hackers Association

September 2024

Choral: Object-Oriented Choreographic Programming

- A *choreography* is a **coordination plan** between many parties - e.g. a block cipher, authentication protocol - like interface for distributed systems
- **Choral** is new PL designed for implementing choreographies, better than previous solutions b/c it has first-class knowledge of "data distributed across roles"
- Choral compiler translates your choreography into Java implementations of each role of the choreography - first-class choreography knowledge allows for unique optimization, therefore efficient distributed algorithms
- Example implementations: Mergesort, SSO, transmitting healthcare IoT vitals (PHI)

Choral: Object-Oriented Choreographic Programming

Hello roles. All values in Choral are distributed over one or more roles, using the @-notation seen in Section 1. The degenerate case of values involving one role allows Choral to reuse existing Java classes and interfaces, lifted mechanically to Choral types and made available to Choral code. For example, the literal `"Hello from A"@A` is a string value `"Hello from A"` *located at role* `A`. Code involving different roles can be freely mixed in Choral, as in the following snippet.

```
class HelloRoles@(A,B) {
 public static void sayHello() {
  String@A a = "Hello from A"@A;
  String@B b = "Hello from B"@B;
  System@A.out.println(a);
  System@B.out.println(b);
 }
}
```

Choral Code

Boosting Compiler Testing by Injecting Real-World Code

- $\exists$ *random program generators* that we use to test optimizing compilers, but they kinda suck - difficult to generate programs that make rich use of more complex syntactic/semantic features of language
- Idea: let's extract code from real, production-level projects (web servers, operating systems, desktop apps, etc.) and use that as tests for our compilers
- This is tough - identifiers have scopes and C has DMA, so data dependencies are hard to track
- Solution: extract individual functions, call them within some "seed" code, and use dynamic execution info to provide args to the function calls
- Allows for testing compilers with human-written code on a large scale

Daedalus: Safer Document Parsing

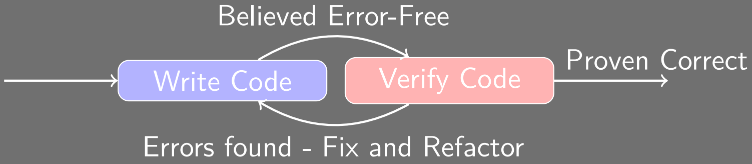
- Parsers are the target of un-countably many vulnerabilities because **a)** we write them all the time (foundation of computer science as a practice) and **b)** humans suck **so bad** at writing parsers
- If we can't be trusted to write safe parsers, there should be a tool that we can use to write them - enter **Daedalus**, a DSL for writing safe binary file parsers
- Generates parsers in C++ and Haskell
- Paper provides 20 implementations to common binary formats, including a PDF implementation that has been red-teamed by the Adobe foundation and is shown to be significantly less buggy than existing parsers!
- Paper also provides static analysis tools that can *detect potential code injection sectors of your binary format*
- Start using this tool now!

RefinedRust: A Type System for High-Assurance Verification of Rust Programs

- Rust has a rich type system that makes guarantees about memory safety, although this can be turned off in `unsafe` mode
- Many verification tools utilize Rust's type system to give users ability to write proofs about their code, but they do not support `unsafe` mode
- **RefinedRust** is a *refinement type system*, verified secure in Coq, that allows for semi-automated proofs about arbitrary Rust code - including `unsafe` mode
- A refinement type system is halfway to *dependent typing* - it allows for applying arbitrary logical predicates to data types (e.g. "int that is greater than 0" rather than just "int")
- Paper provides verified `Vec` implementation that uses unsafe pointer manipulation practices

Live Verification in an Interactive Proof Assistant

- Standard formal verification workflow:



- This can cumbersome for large code because errors found are complex
- Paper provides what is essentially a REPL for verified C software development - write a line of code, get immediate feedback from Bedrock2 (Coq-based C verification tool) about symbolic state of program and provide proof tactics in C comments to modify proof state

Input-Relational Verification of Deep Neural Networks

- We want to verify *input-relational* properties (a property that states something about the output of a corresponding input) of DNNs - difficult because that reasoning requires thinking about many runs of same DNN
- Key insight - new representation of input-relational problem, *DiffPoly*, that allows for computation of differences in outputs between a pair of DNN executions
- Paper also provides *RaVeN*, a verification framework that utilizes DiffPoly to find data dependencies between layers (better than previous tools) which scales up to describing the behavior of the entire network

The Functional Essence of Imperative Binary Search Trees

- When you learned tree algorithms, you saw stuff like

function $\textsc{BinarySearch}(A, n, x)$
    $low \leftarrow 1$
    $high \leftarrow n$
    while $low \leq high$ **do**
        $mid \leftarrow \lfloor (low + high)/2 \rfloor$
        if $A[mid] = x$ **then**
            return mid
        else if $A[mid] < x$ **then**
            $low \leftarrow mid + 1$
        else
            $high \leftarrow mid - 1$
        end if
    end while
    return `NOT FOUND`
end function

The Functional Essence of Imperative Binary Search Trees

- Key observation: this is an imperative algorithm - they are difficult to verify because they require difficult intuition about partial correctness of algorithm w.r.t. subtrees
- We should have **performant** functional counterparts to imperative tree algorithms - they are much easier to formally verify
- That's what this paper does! It provides novel functional algorithms for accessing and inserting in a *restructuring* BST (move-to-root, splay, zip, etc.)
- Provided algorithms are formally verified in Coq via the Iris platform (used for verifying concurrent code) and are shown to be nearly as performant as standard, imperative C implementations

Qubit Recycling Revisited

- Quantum circuits pass some number of *qubits* (bits that can be in a superposition of 0 and 1) through *gates* that inplace-modify the states of said qubits
- Qubits are really hard to create and maintain, and the more logical qubits you want, the more error-correcting qubits you need - ergo, reducing the *width* (number of qubits) of a quantum circuit is useful
- This paper revisits qubit *recycling*, which reuses qubits that have already been discarded/deallocated/measured - seems simple, but $O(n!)$ possible ways to reuse qubits!
- New solution for recycling uses a novel comparison to a known matrix-triangularization problem, proves that it's NP-Hard, gives a solver that finds the optimal solution for the majority of circuits tested, and is formally verified in Coq to maintain circuit semantics! Wow!

Thank you!

- Questions?
- Slides available on my website: https://seashell.charles.systems/teaching
- CSG CTF is seeking corporate sponsorship - we were the 8th-best university in the country last semester!
- Talk to me if you've ever worked on/reasoned about voting software