Facilitating Verified Software Development

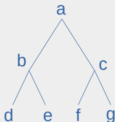
Christine Rizkallah
The University of Melbourne

Research Seminar, FMoz, Brisbane, Australia

05 June 2026

Code and Data Refinement

	Verification		**Implementation**
Data	Algebraic Datatypes		Bits and Bytes in Memory
Code	Mathematical Functions		Imperative Instructions



a	b	c	d	e	f	g

Approach

Aim: Create a method to **facilitate verification** without compromising on **trust** or **efficiency**.

To reduce the manual effort:

- functional languages increase productivity, ease verification
- type systems can auto. enforce safety, security properties
- certifying compilers automate large portion of verification

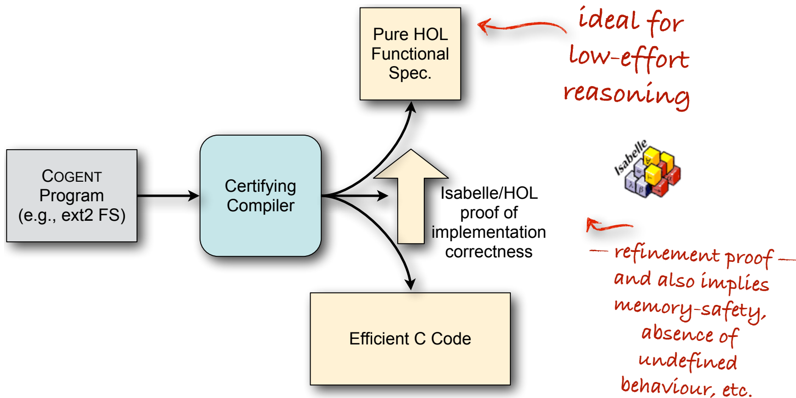
past: filesystems, device drivers
future: network protocols, FFIs, APIs

The Cogent Language

Trustworthy Filesystems

- motivation: filesystems are too important to remain unverified
- problem: posed as **grand challenge** [Freitas et al.'08]
- issue: many huge filesystems (each $\approx 5k$ loc, Linux has ≈ 50)
- goal: elegant method to reduce effort of verifying filesystems
- solution: created **Cogent**: a restricted **functional language** with a **linear type system** and a **certifying compiler** that co-generates code and proofs

Cogent's Certifying Compiler



Cogent: Verification Results in a Nutshell

- built **efficient** Cogent filesystems (ext2 & BilbyFS)
  - linearly typed variables are used exactly once
  - in-place memory updates; no garbage collector
  - generated C within 10% overhead of hand-written C
  - each filesystem $\approx 4k$ Cogent loc, plus $\approx 2k$ C loc
- Cogent **drastically reduced cost** of verification
  - reasoning about HOL functional spec. instead of directly on C.

seL4	$\approx$	1.65 person months per 100 C loc
filesystems	$\approx$	0.38 person months per 100 Cogent loc

  - thanks to linear types: automatic proof of memory safety

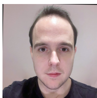
Cogent: Verification Results in a Nutshell

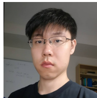
- built **efficient** Cogent filesystems (ext2 & BilbyFS)
  - linearly typed variables are used exactly once
  - in-place memory updates; no garbage collector
  - generated C within 10% overhead of hand-written C
  - each filesystem $\approx 4k$ Cogent loc, plus $\approx 2k$ C loc
- Cogent **drastically reduced cost** of verification
  - reasoning about HOL functional spec. instead of directly on C.

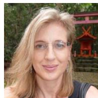
seL4	$\approx$	1.65 person months per 100 C loc
filesystems	$\approx$	0.38 person months per 100 Cogent loc

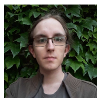
  - thanks to linear types: automatic proof of memory safety

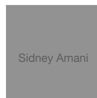
Among Others, Cogent Contributers


Liam O'Connor
University of Edinburgh


Zilin Chen
Ghost Autonomy


Gabriele Keller
Uttrecht University


Vincent Jackson
University of Melbourne

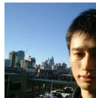


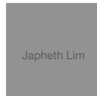
Sidney Amani

Louis Cheung
University of Melbourne

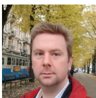
Toby Murray
University of Melbourne

Amos Robinson
ANU


Yutaka Nagashima
Huawei



Japheth Lim


Thomas Sewell
University of Cambridge

Gernot Heiser
UNSW

Gerwin Klein
Proof Craft

Partha Susarla

Alex Hixon

Type Mismatch Problem (Cogent, C)

- in systems, these types are used:
  - bitfields
  - padding and alignment
- none of these can be expressed by a native Cogent type
- old solution:
  - defines an abstract type for the native C type,
  - a logically equivalent Cogent type,
  - and a pair of conversion functions back and forth.
- drawbacks: error-prone, inefficient, hard to verify
- can we do better?

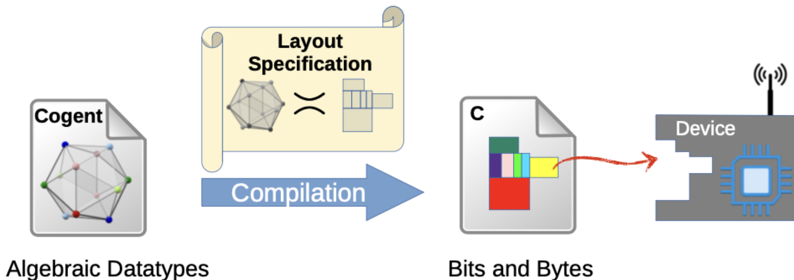
Yes, we can

1. Dargent: shift towards more Cogent and less C
2. Verified FFI: compose verified Cogent and C

Dargent: Verified Data Refinement

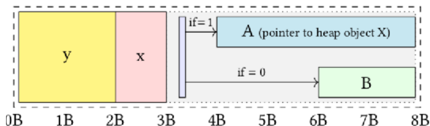
Trustworthy Device Drivers

Dargent is a **declarative data layout specification language** that enables the automation of **verified data refinement** from **algebraic types** to their **memory layouts** down to the bit level.



Dargent Example

$\left\{ x : \mathsf{U8},\ y : \mathsf{U16},\ z : \left\langle \mathrm{A}\ X \mid \mathrm{B}\ \mathsf{U16} \right\rangle \right\}$

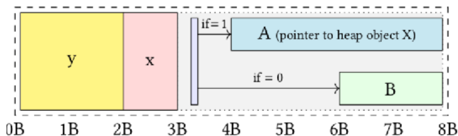


Dargent Example

{ x : U8 , y : U16 , z : ⟨ A X | B U16 ⟩ }

layout *Example* = record { y : 2B at 0b
, x : 8b at 2B
, z : *Nested* at 3B }

layout *Nested* = variant (1b at 2b)
{ A(1) : 32b at 1B
, B(0) : 16b at 24b }

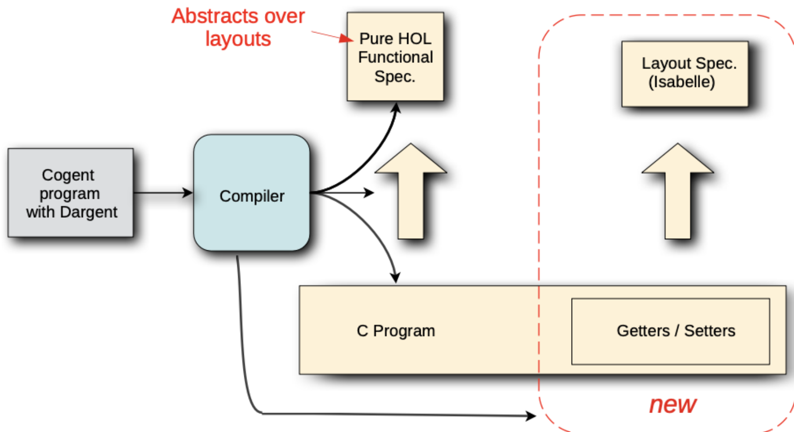


Dargent Compilation

Cogent	C (without Dargent)	C (with Dargent)
`{ a : U32, ... }`	`struct { int a; ... }`	`int[n]`
`r.a`	`r.a`	`get_a(r)`
`r.a = 4`	`r.a = 4`	`set_a(r,a)`

The getter and setter
- are generated by the compiler
- as specified by the layout
- allow coding using the algebraic type

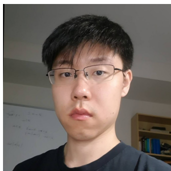
Cogent with Dargent: Certifying Compiler



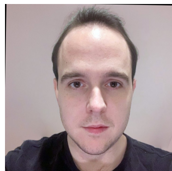
Dargent Results in a Nutshell

- Dargent is a data layout specification language for **verifying data layout refinement** [POPL'23]
- eliminates the need for serialisation and deserialisation code
- supports **polymorphic layouts** and **endianness annotations**
- the certifying compiler generates:
  1. getters and setters to lay out and access types as specified
  2. an Isabelle/HOL proof that they respect the Dargent layout
- implementation and verification still operate on algebraic types
- verified a **timer driver** and a **power control system**
- Dargent trivialised both implementation and verification

Among Others, Dargent Contributors


Zilin Chen
NTU Singapore

Ambroise Lafont
École Polytechnique


Liam O'Connor
ANU

Gabriele Keller
Utrecht University

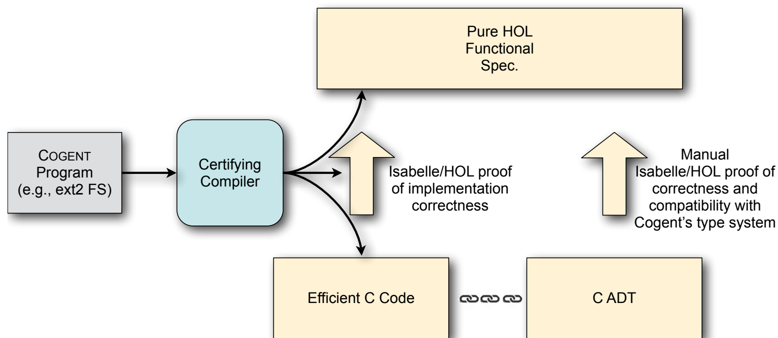
Craig McLaughlin
Category Labs

Vincent Jackson
UNSW

Verified Foreign Function Interface (Cogent, C)

- Cogent programs are part of a larger C system
- they are connected using a foreign function interface (FFI)
- recall aim: only **small** part in C (manual verification)
- currently, most of the C parts consist of ADTs, ADT functions
- challenge: bridge between type-safe Cogent and unsafe C

Cogent's Certifying Compiler and ADTs



- aim: horizontally compose Cogent and foreign C code
- manually verified a C word array implementation and
- result: verify Cogent-C system using FFI [CPP'22]

Composing Safety

- uniqueness types (Cogent): safety by design
- separation logic (C): frame rule for local reasoning
- condition: C must respect Cogent's uniqueness inv.

Uniqueness is Separation: Mechanised Results

- Separation logic eases discharging Cogent's FFI obligations [APLAS'25]
- contributions:
  1. a **separation-logic characterisation** of the memory-safety invariants enforced by Cogent's uniqueness type system
     - the FC Triple implies all three frame conditions (proved)
     - a set of derived rules to prove FC Triples for $\mathcal{L}_{IMP}$ programs
  2. a **Rocq**: the first formalisation of SL using nominal techniques
- consequence: existing C verification tools based on SL can be **directly reused** to discharge Cogent's FFI obligations
- type-system and proof-based guarantees in a single logic

Big Picture: Verified FFIs and API, Concurrency

Verified APIs and FFIs [CPP'22,APLAS'26 ...]

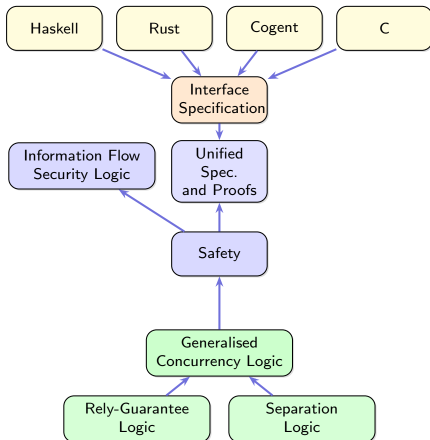
Cross-language composition today requires large volume of **hand-written FFI proofs**.

Goal: auto-generate FFI proofs from interface specs so verified components compose without trust gaps.

Concurrency [ITP'24, ...]

Shared state, races, and interference make concurrent code notoriously hard to verify.

Goal: a composable rely-guarantee + separation logic framework that scales to OS-level code without re-doing proofs from scratch.



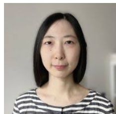
Current PhDs and Recent Graduates

Dr. Louis Cheung
Information Retrieval

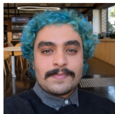
Dr. Zoey Chen
Cost Models

Vincent Jackson
Concurrency

Selene Linares
Memory Safety

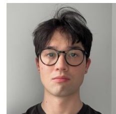


Xing Li
Cost Models



Sadra B. Tork
Binary Analysis

Jack Ho
AI for Prog. Repair



Jeremy Lindsay
Proof Refactoring

Summary

Aim: reduce the manual effort required to verify software without compromising performance or trust.

Cogent uniqueness types, certifying compiler (filesystems) [ICFP'16, APLAS'16, ITP'16, JFP'21]

Dargent declarative data layouts, refinement (device drivers) [POPL'23]

Verified FFI separation logic to discharge uniqueness cond. [CPP'22, ASPLOS'25]

Cost Analysis of functional code including lazy programs [ITP'25, ESOP'26]

Concurrency RGSep for concurrent code [ITP'24]

Security SecRGSep for extending safety to IFC

Summary

Aim: reduce the manual effort required to verify software without compromising performance or trust.

Cogent uniqueness types, certifying compiler (filesystems) [ICFP'16, APLAS'16, ITP'16, JFP'21]

Dargent declarative data layouts, refinement (device drivers) [POPL'23]

Verified FFI separation logic to discharge uniqueness cond. [CPP'22, ASPLOS'25]

Cost Analysis of functional code including lazy programs [ITP'25, ESOP'26]

Concurrency RGSep for concurrent code [ITP'24]

Security SecRGSep for extending safety to IFC

Thanks for listening.