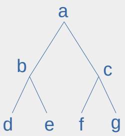
DARGENT: A Silver Bullet for Verified Data Layout Refinement

Zilin Chen   Ambroise Lafont   Liam O'Connor
Gabriele Keller   Craig McLaughlin   Vincent Jackson
Christine Rizkallah

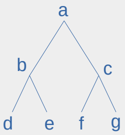
18^{th} January 2023 (POPL)

Developing High-Assurance Systems

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

Developing High-Assurance Systems

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



Cogent

Cogent is a **programming language** and **certifying compiler** for building verified low-level systems

Cogent

Verification

Purely functional

Uniqueness types

Type safe (implies memory safe)

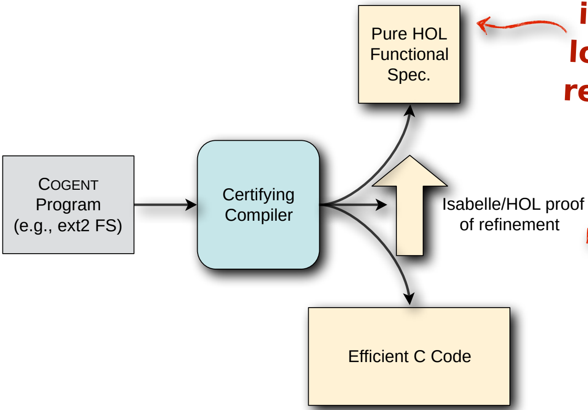
Low-level systems

C Foreign Function Interface

No garbage collection

Destructive updates

Cogent's Certifying Compiler



ideal for low-effort reasoning

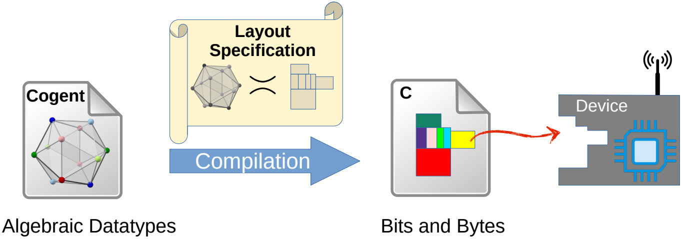
— refinement proof — also implies memory-safety, absence of undefined behaviour, etc.

What about datatypes and their memory layouts?

Dargent

Dargent is a **data layout specification language** with **verified data refinement** from **algebraic types** to their **memory layouts** down to the bit level

Dargent


Layout
Specification
Cogent
Compilation
C
Device
Algebraic Datatypes
Bits and Bytes

+

Compilation with Dargent Specifications

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)`

Compilation with Dargent Specifications

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

Example: Power Control System

"PWR_CR1.VOS = scale"

C: error-prone code

```
reg32 = PWR_CR1 & ~(PWR_CR1_VOS_MASK << PWR_CR1_VOS_SHIFT);
reg32 |= (scale & PWR_CR1_VOS_MASK) << PWR_CR1_VOS_SHIFT;
PWR_CR1 = reg32;
```

Cogent + Dargent

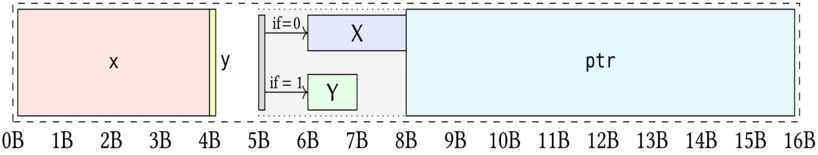
```
pwr_cr1 { vos = scale }
```

+ **layout specification** of the device register

An Example of Dargent

type `Example` = {
  `struct` : #{`x` : `U32`, `y` : `Bool`},
  `ptr` : {. . .},
  `sum` : ⟨`X` `U16` | `Y` `U8`⟩
}

layout `ExampleLayout` = record {
  `struct` : record {x : 4B, y : 1b},
  ptr : pointer at 8B,
  sum : variant (1b)
    {X(0) : 2B at 1B, Y(1) : 1B at 1B} at 5B
}



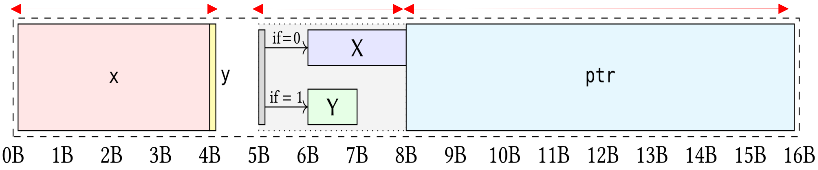
An Example of Dargent

record
pointer
variant

```
type Example = {
    struct : #{x : U32, y : Bool},
    ptr : {...},
    sum : ⟨X U16 | Y U8⟩
}
```

```
layout ExampleLayout = record {
    struct : record {x : 4B, y : 1b},
    ptr : pointer at 8B,
    sum : variant (1b)
        {X(0) : 2B at 1B, Y(1) : 1B at 1B} at 5B
}
```

record's layout
pointer's layout
variant's layout

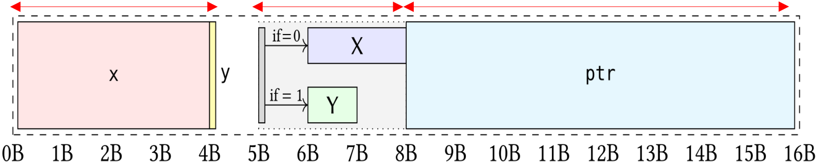


An Example of Dargent

type Example = {
record   struct : #{x : U32, y : Bool},
pointer   ptr : {. . .},
variant   sum : ⟨X U16 | Y U8⟩
}

layout ExampleLayout = record {
struct : record {x : 4B, y : 1b}, record's layout
ptr : pointer at 8B, pointer's layout
sum : variant (1b) variant's layout
{X(0) : 2B at 1B, Y(1) : 1B at 1B} at 5B
}

layout size in bits & Bytes



An Example of Dargent

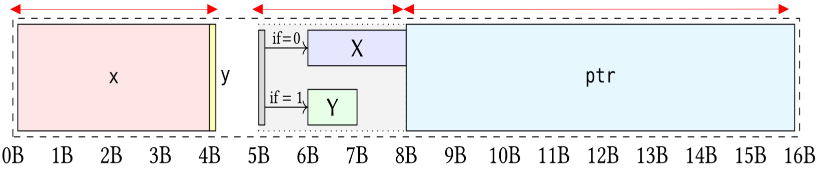
record
pointer
variant

```
type Example = {
    struct : #{x : U32, y : Bool},
    ptr : {...},
    sum : ⟨X U16 | Y U8⟩
}
```

```
layout ExampleLayout = record {
    struct : record {x : 4B, y : 1b},
    ptr : pointer at 8B,
    sum : variant (1b)
        {X(0) : 2B at 1B, Y(1) : 1B at 1B} at 5B
}
```

record's layout
pointer's layout
variant's layout

layout size in bits & Bytes
offset



An Example of Dargent

```
type Example = {
        struct : #{x : U32, y : Bool},
        ptr : {...},
        sum : ⟨X U16 | Y U8⟩
}
```

record — pointer — variant

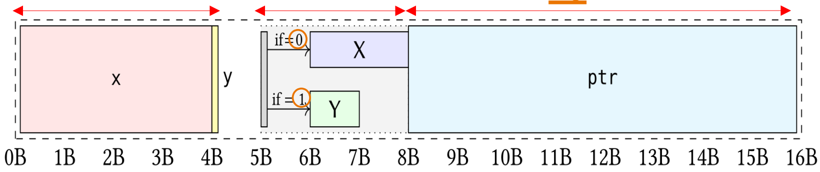
```
layout ExampleLayout = record {
    struct : record {x : 4B, y : 1b},
    ptr : pointer at 8B,
    sum : variant (1b)
        {X(0) : 2B at 1B, Y(1) : 1B at 1B} at 5B
}
```

record's layout — pointer's layout — variant's layout

layout size in bits & Bytes

offset

tag



Bit Fields

C

```
struct can_id {
  uint32_t id:29;
  uint32_t exide:1;
  uint32_t rtr:1;
  uint32_t err:1;
};
```

Bit Fields

C

```
struct can_id {
  uint32_t id:29;
  uint32_t exide:1;
  uint32_t rtr:1;
  uint32_t err:1;
};
```

Cogent

```
type CanId = {
  id : U29,
  exide : Bool,
  rtr : Bool,
  err : Bool
}
```

Dargent

```
layout record {
  id : 29b,
  exide : 1b,
  rtr : 1b,
  err : 1b
}
```

Assigning layouts

`{ a : U8 , ... }` layout $\overbrace{\ell}^{\text{a layout spec.}}$

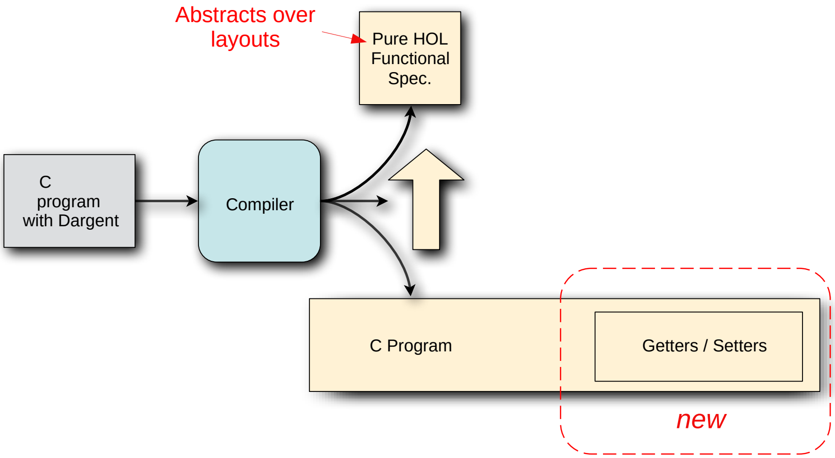
Assigning layouts

$$\texttt{\{ a : U8 , ... \}}\ \textcolor{blue}{\text{layout}}\ \overbrace{\textcolor{blue}{\ell}}^{\text{a layout spec.}}$$

Requirements:

- ℓ **wf**
- $\{\ \texttt{a : U8 , ...}\ \} \asymp \ell$

Cogent + Dargent Specifications



Relating Records ($C \sqsubseteq$ HOL)

Cogent + Dargent		C
`{ a : U8, ... }` layout ℓ	compiles to $\longrightarrow$	`int[n]`
`r.a`		`get_a(r)`
`r.a = 4`		`set_a(r, 4)`

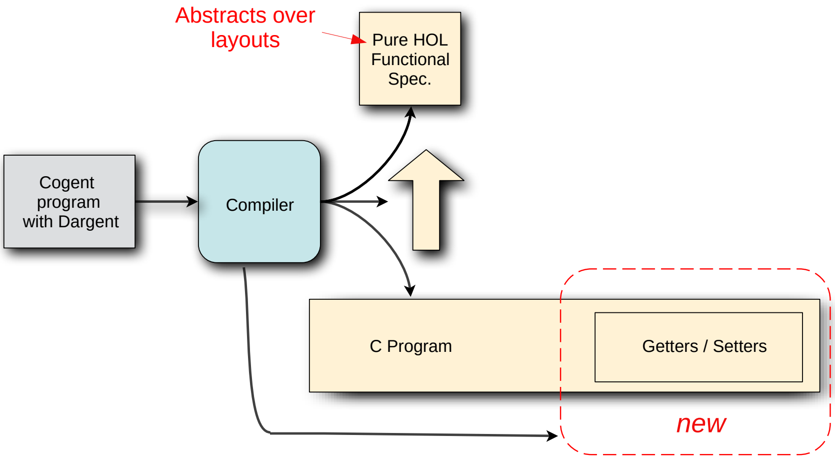
Required properties of getters and setters ($C \sqsubseteq$ HOL)?

Auxiliary lemmas (C $\sqsubseteq$ HOL)

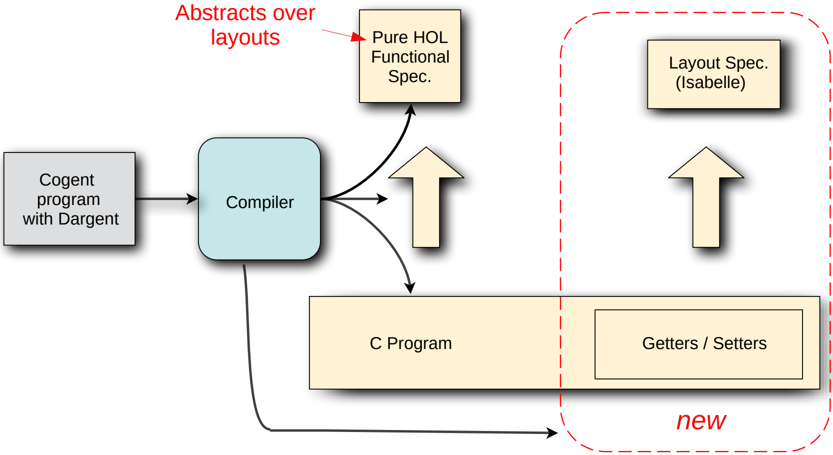
- $\texttt{get\_a}(\texttt{set\_a}(r, a)) \sim a$
- $\texttt{get\_a}(\texttt{set\_b}(r, b)) = \texttt{get\_a}(r)$

Compatibility between getters/setters and layouts?

Cogent + Dargent Specifications



Cogent + Dargent Specifications



Compatibility with layouts

- `set_a` does not flip any bit outside $\overbrace{\ell_a}^{\text{layout spec for the field } a}$
- `get_a` complies with ℓ_a

Compliance with the layout

$$\forall arr,\ \texttt{get\_a}(arr) \sim \textbf{getter}(\ell_a, arr, \tau_a)$$

$$\textbf{getter} : \mathit{dataArray} \to \mathit{layout} \to \mathit{cogentType} \to \mathit{cogentValue}$$

(implemented once and for all in Isabelle)

Related Work

Data Description Languages: PackTypes, Protege, DataScript, ...

Different purposes:

- DDLs = external domain-specific languages

  Goal: generate encoders/decoders

- DARGENT = PL extension (COGENT)

  Goal: customise low-level behaviour of compiled types

Layouts in type systems: LoCaL, SHAPES, hobbit

Relevant DARGENT features:

✓ Formal guarantees

× No recursive types, no arrays (WIP)

Conclusion

Dargent: a language for specifying layouts of algebraic types

- Declaratively specify their desired layouts in Dargent
- 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 manual verification still operate on algebraic types

Examples: Systems applications

- Verified timer driver
- Power control system