Dargent: A Silver Bullet for Verified Data Layout Refinement
(POPL 2023)

Chen-Lafont-O'Connor-Keller-McLaughlin-Jackson-Rizkallah

Dresden, 2nd June 2023

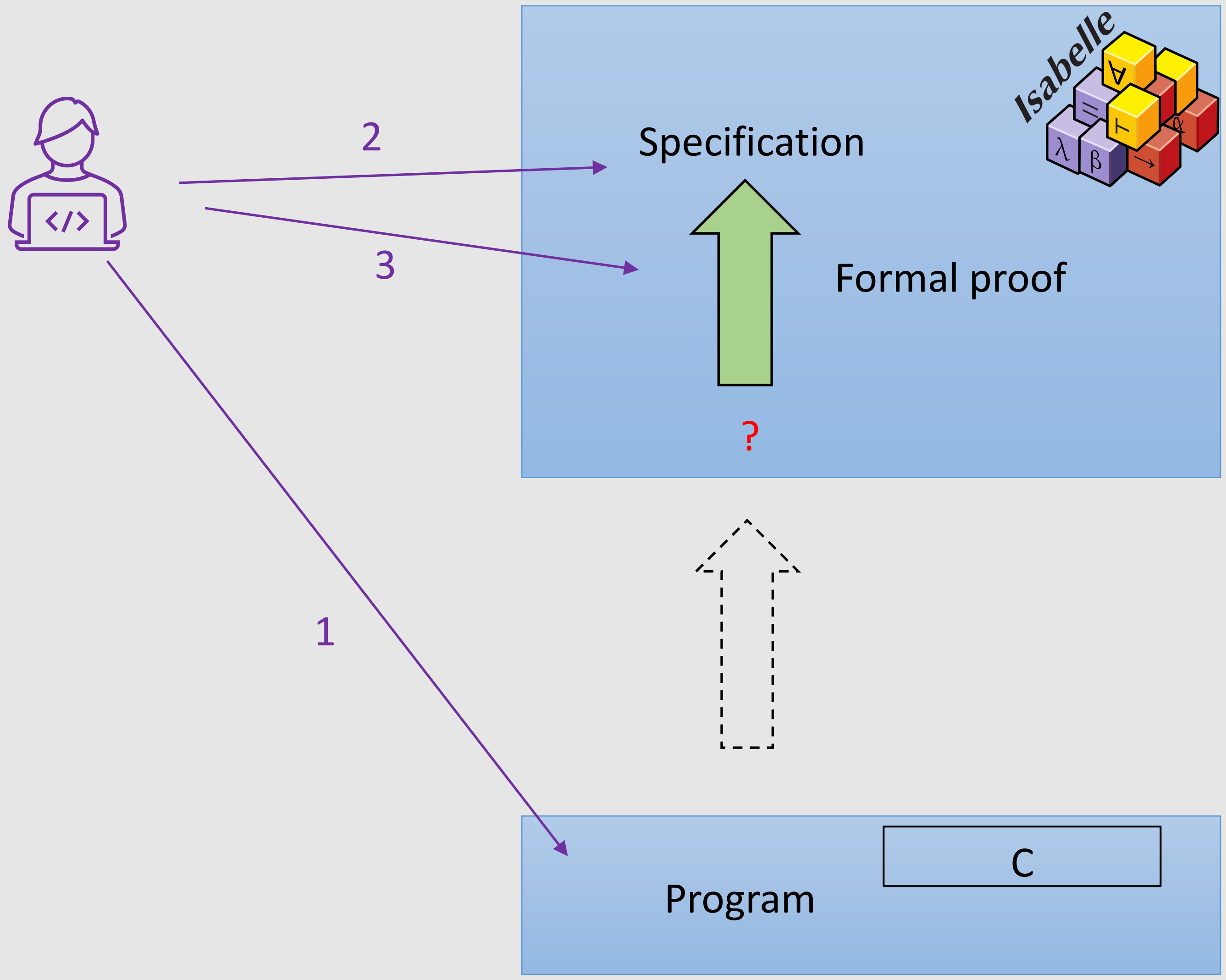
Context

Security. Performance. Proof.

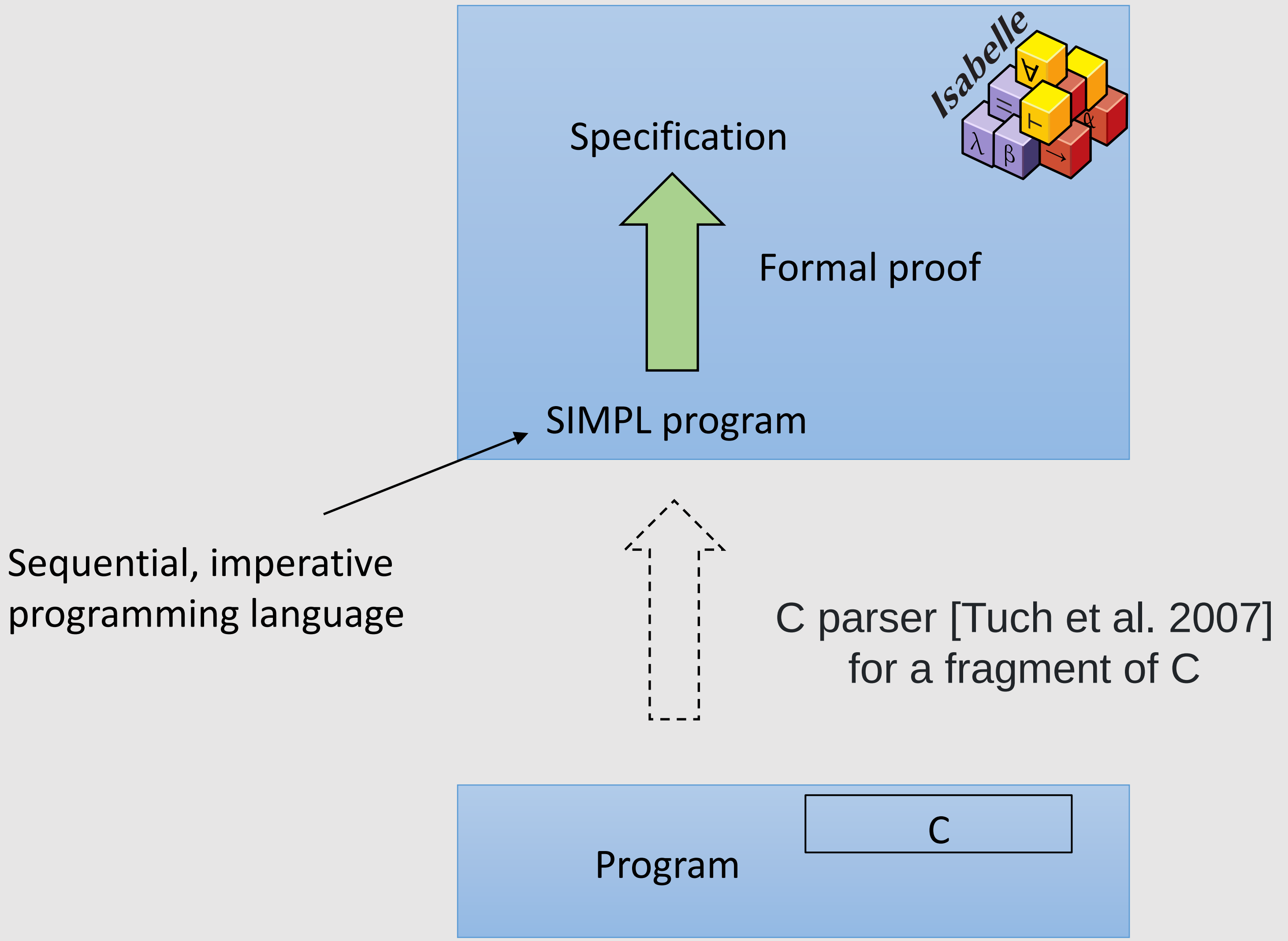
The world's most secure operating system kernel.

Used in
Autonomous vehicles

How does the sel4 team reason on C code?



How does the sel4 team reason on C code?



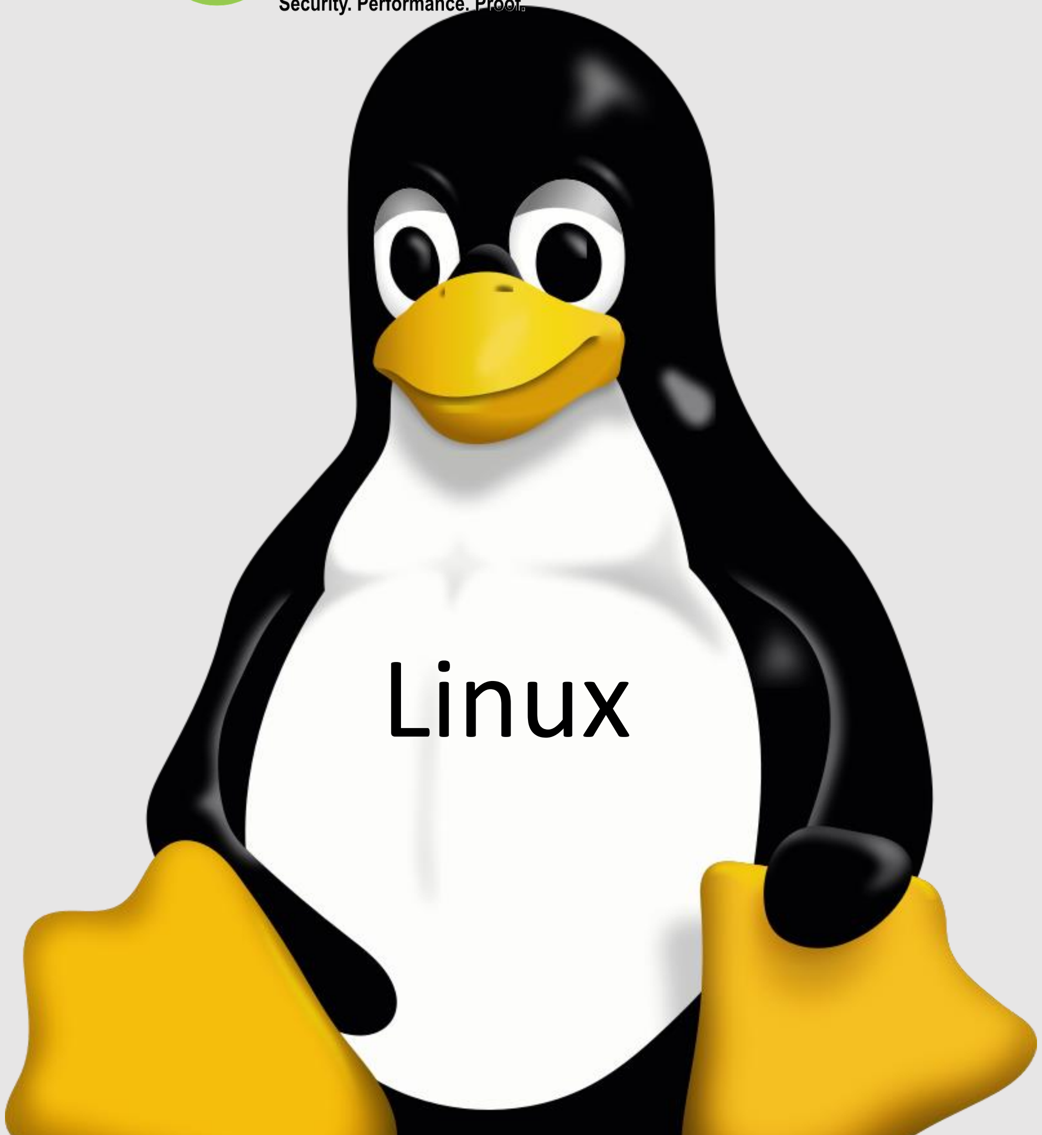
seL4
Security. Performance. Proof.

Micro-kernel: 10k lines of C

PROOF

200k lines

Isabelle



30M lines of C

File systems, device drivers, …

How to ease systems code certification?

A simpler embedding of C

Specification

Formal proof

Monadic program

AutoCorres [Greenaway et al. 2012]

computes *monadic* semantics
for a subfragment of the C parser

Program

C

The monadic semantics of AutoCorres

C

Type A

$f : A \rightarrow B$

AutoCorres

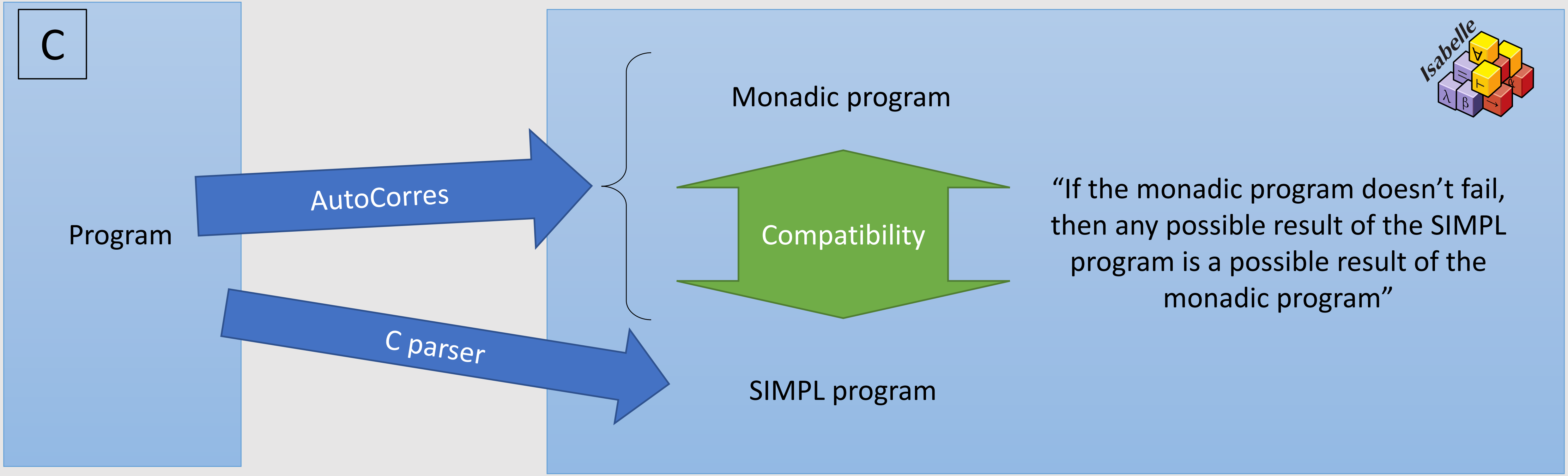
Type A_m

$$f_m : \underbrace{A_m \rightsquigarrow B_m}_{A_m \times state \;\rightarrow\; (B_m \times state)\; option}$$

Effects :

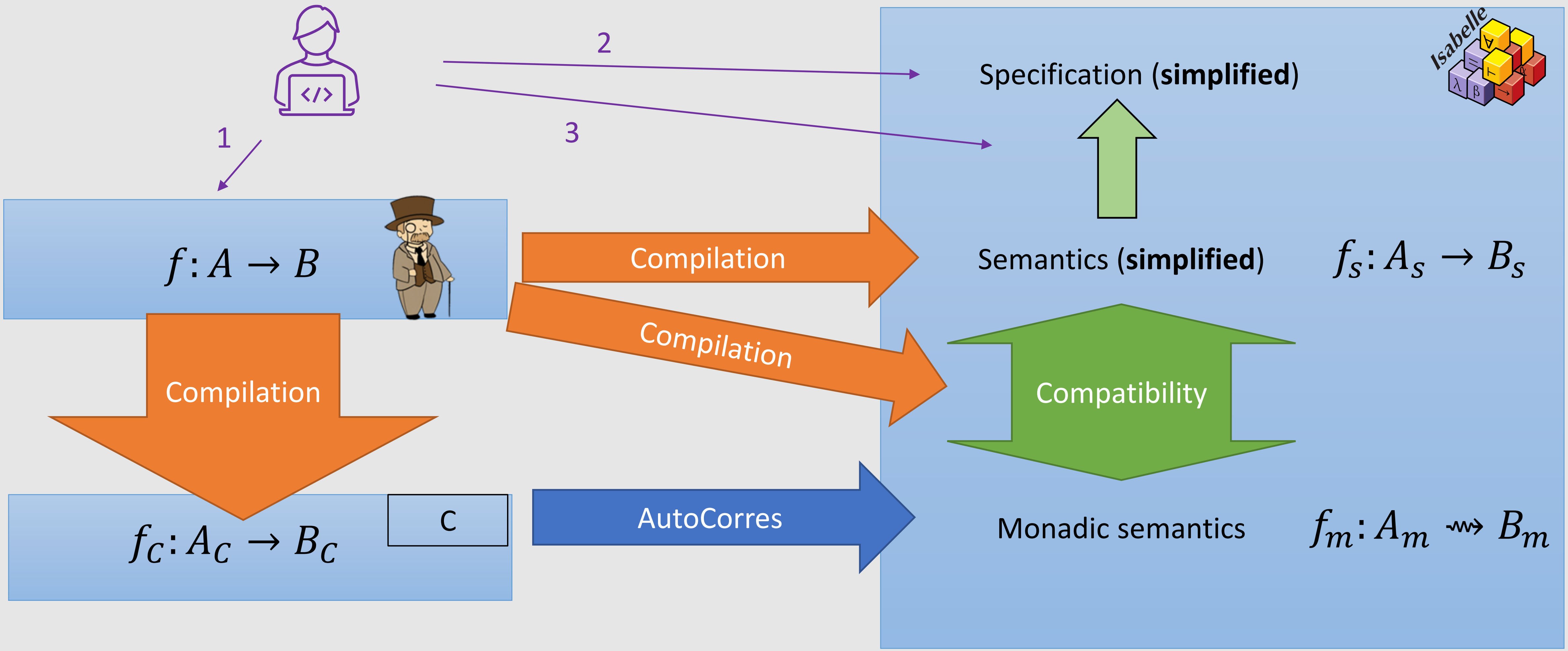
- Memory state
- Error
- *(non determinism)*

Justifying the monadic semantics



Cogent: Simplified semantics

A total functional language to describe a ***safe*** subfragment of AutoCorres, with **simplified semantics**



Comparing semantics

Monadic semantics	**Simplified semantics**
Type T_m	Type T_s
$f_m : A_m \rightsquigarrow B_m$	$f_s : A_s \to B_s$

Compatibility between the two semantics
Relation $T_r \subset state \times T_m \times T_s$
(f_m, f_s) is *compatible* with (A_r, B_r)

« (f_m, f_s) maps related values to related values »
(in particular, f_m doesn't fail)

Simplifications :

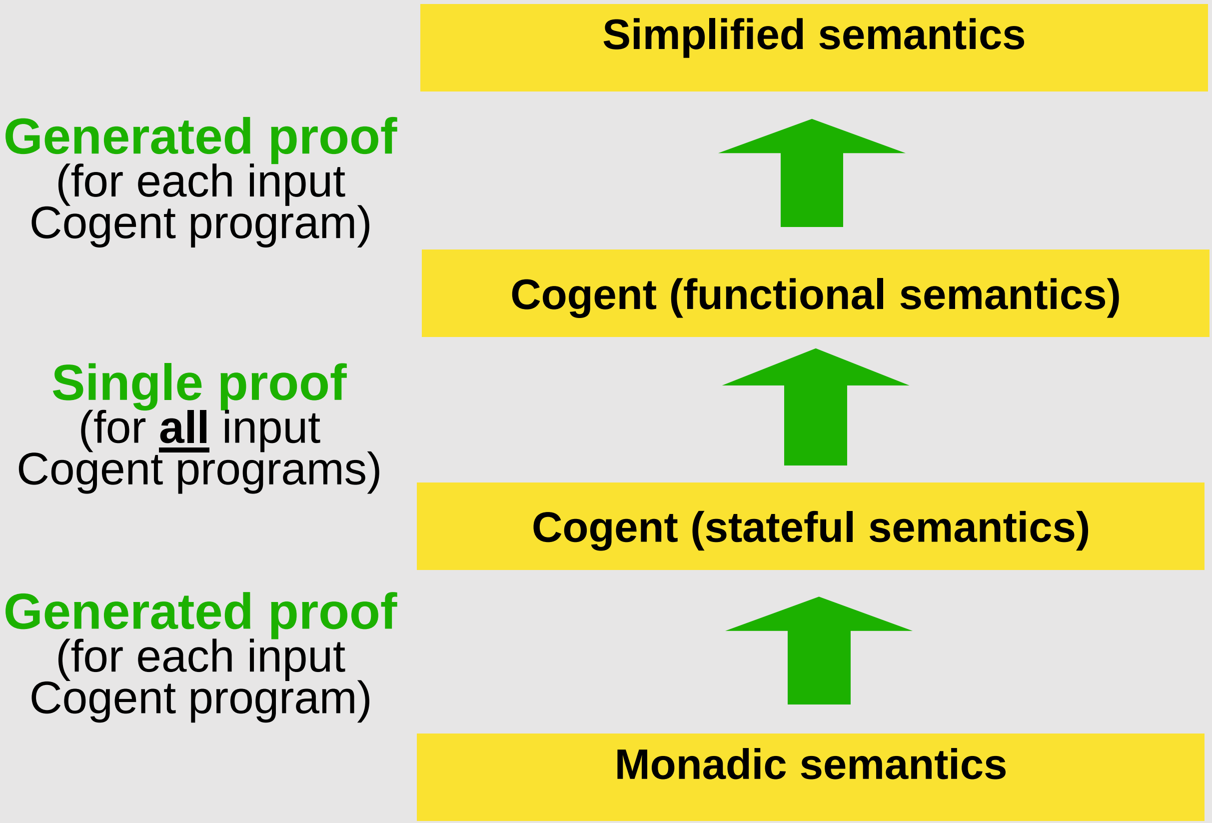
- No pointer / memory

$T_m = A_m{}^*$	$T_s = A_s$	$T_r = \{(memory, p, memory(p))\}$

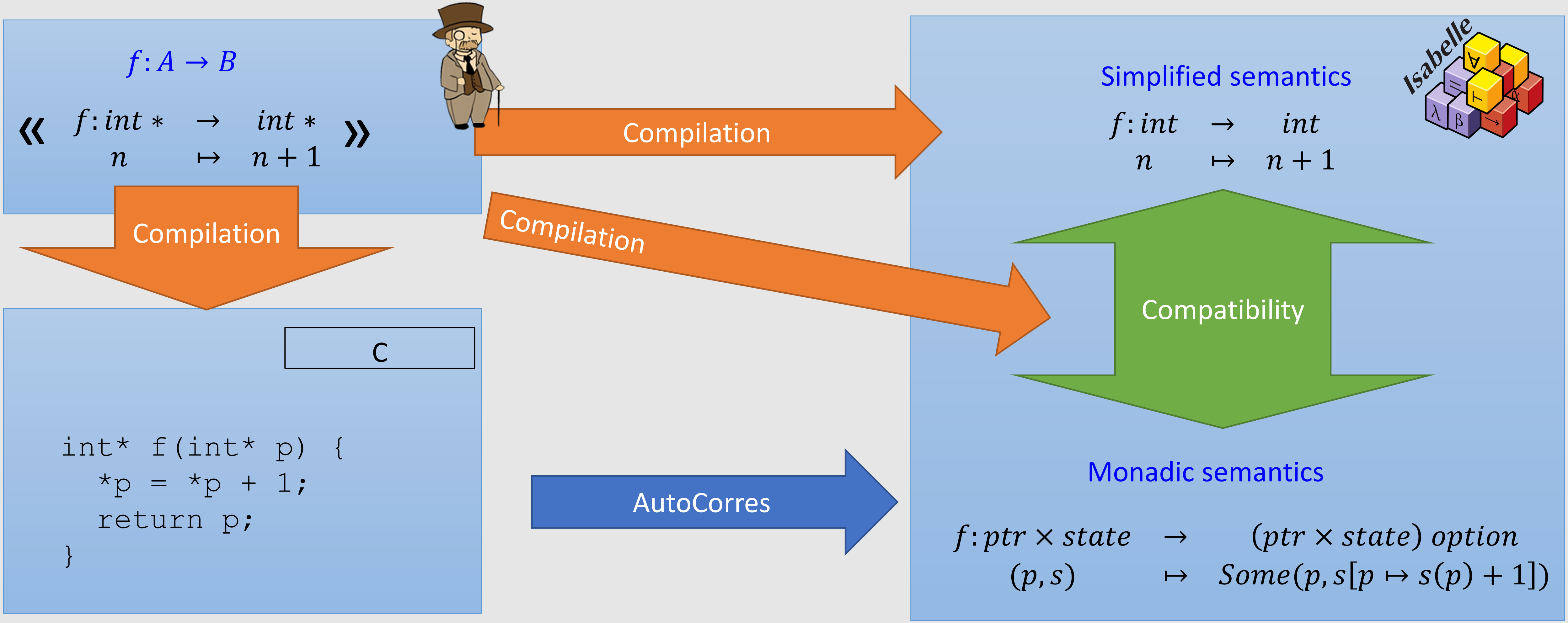
- Pure functional semantics

$f_m : int \rightsquigarrow int$	$f_s : int \to int$	$f_m(s, n) = Some(s', f_s(n))$

Compatibility, under the hood



Example


$f: A \to B$
« $f: int* \to int*$, $n \mapsto n+1$ »
Compilation
Simplified semantics
$f: int \to int$
$n \mapsto n+1$
Compilation
Compilation
Compatibility
C
int* f(int* p) {
*p = *p + 1;
return p;
}
AutoCorres
Monadic semantics
$f: ptr \times state \to (ptr \times state)\ option$
$(p, s) \mapsto Some(p, s[p \mapsto s(p) + 1])$
Isabelle

Cogent

A total functional programming language with FFI support

- words (unsigned 8-, 16-, 32- and 64-bit)
- records (structs)
  - e.g. `{f1 : A, f2 : U32}`
- variants (tagged unions)
  - e.g. `<A t1 | B t2>`
- abstract types
- function types
- polymorphic types

Cogent types

Cogent's linear type system

- variables with a linear type must be used exactly once
- Cogent's linear type system:
  - allows generating efficient imperative code with in-place updates
  - ensures memory safety

Limitations

Vast majority of FS-specific code

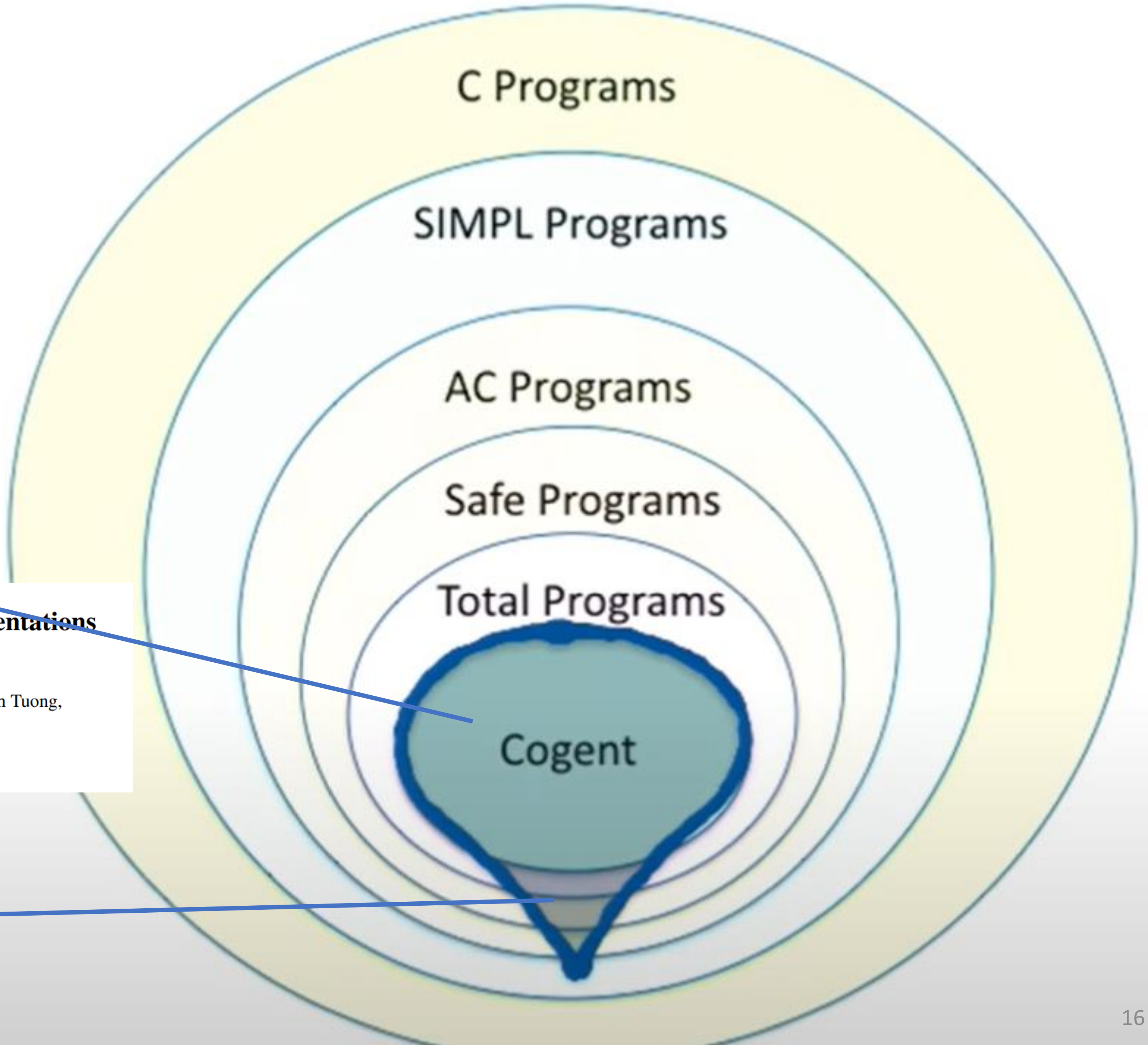
COGENT: Verifying High-Assurance File System Implementations

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb,
Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong,
Gabriele Keller, Toby Murray, Gerwin Klein, Gernot Heiser

Data61 (formerly NICTA) and UNSW, Australia

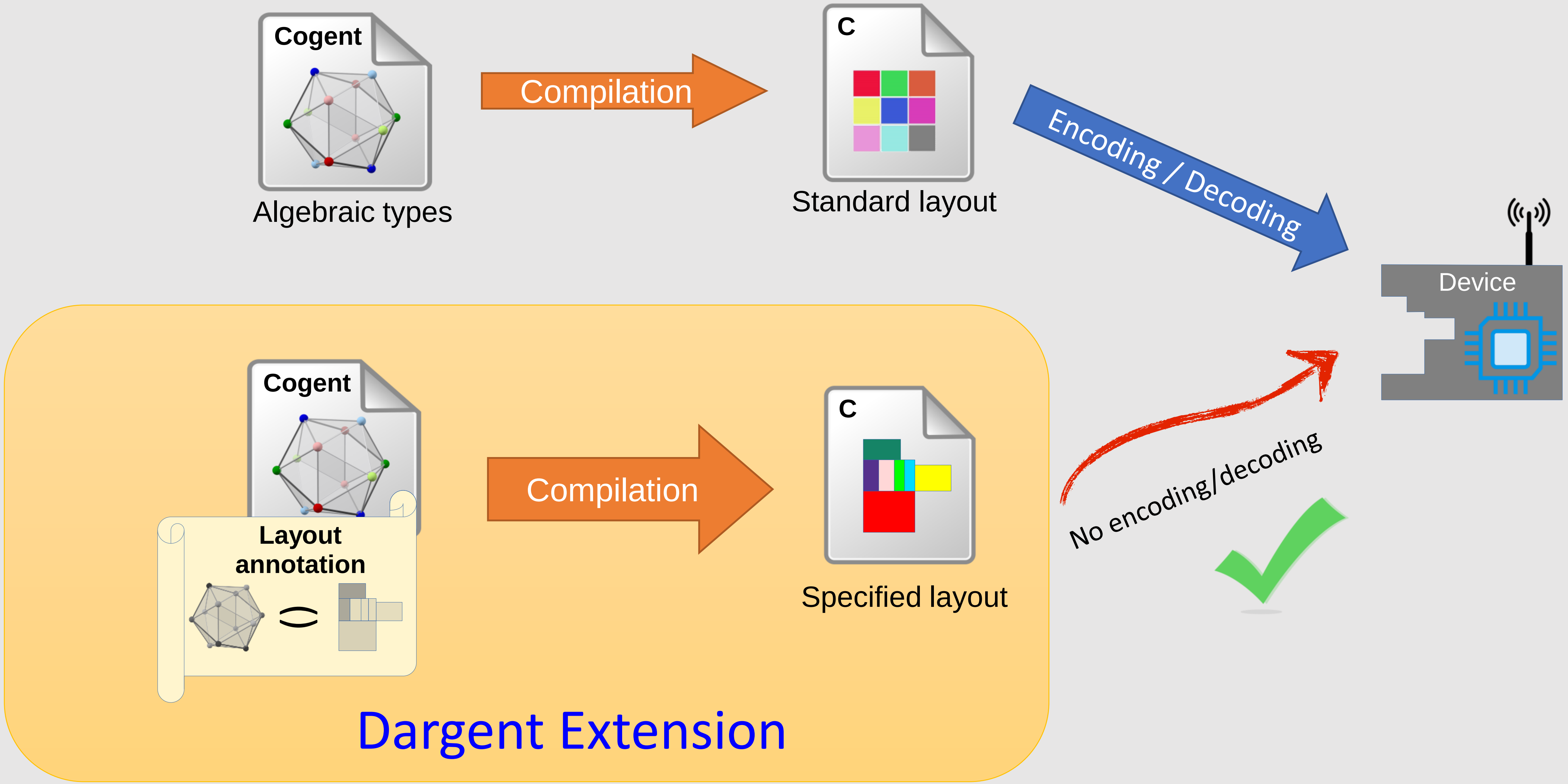
first.last@data61.csiro.au

Loop iterators, data structures, allocators (FFI)



Dargent

Customising the data layouts



Cogent^{+} = Cogent + Dargent

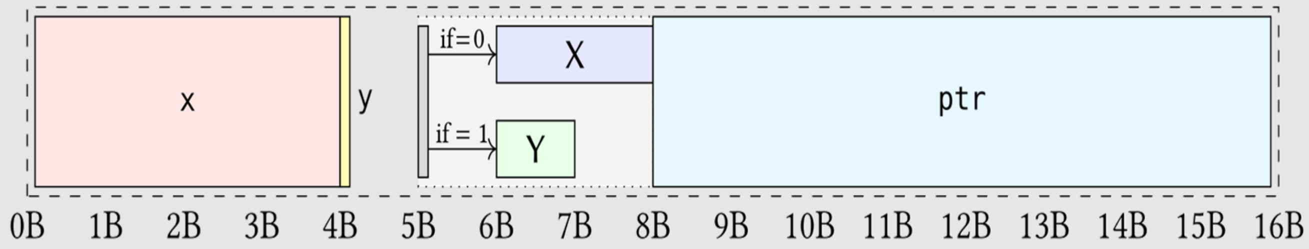
Cogent enriched with the possibility to annotate record types with explicit layouts.

Dedicated syntax

Cogent Type

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

Possible layout



Syntax of Dargent annotations

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

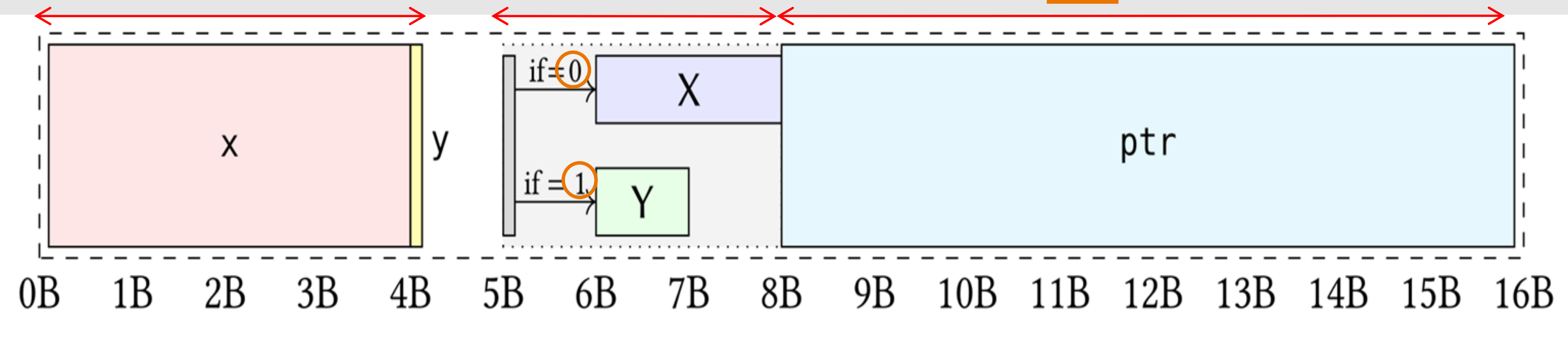
pointer's layout

variant's layout

layout size in bits & Bytes

offset

tag



Simulating Bitfields

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
}
```

Outside the C parser fragment

Some features

- Custom endianness (Little / Big endian) for primitive types
- Layout polymorphism

$$\textbf{type}\ \mathsf{Pair}\ t = \{\mathtt{fst} : t, \mathtt{snd} : t\}$$

$$\textbf{layout}\ \mathsf{LPair}\ l = \mathsf{record}\ \{\mathtt{fst} : l, \mathtt{snd} : l\ \mathsf{at}\ 4\mathsf{B}\}$$

$$\mathit{freePair} : \forall(t, l :\sim t).\ \mathsf{Pair}\ t\ \textbf{layout}\ \mathsf{LPair}\ l \rightarrow ()$$

Matching relation between layouts and types

$$\boxed{\ell \sim \hat{\tau}}$$

$$\frac{\text{bits }(T) = s}{\text{BitRange }(o, s) \sim T}\textsc{PrimTyMatch}$$

$$\begin{aligned} \text{bits }(\mathsf{U}n) &= n \\ \text{bits }(\mathsf{Bool}) &= 1 \end{aligned}$$

$$\frac{M \text{ is the pointer size}}{\text{BitRange }(o, M) \sim \text{pointer}}\textsc{BoxedTyMatch}$$

$$\frac{}{x\{o\} \sim \hat{\tau}}\textsc{VarMatch}$$

$$\frac{}{() \sim ()}\textsc{UnitMatch}$$

$$\frac{\text{for each } i\text{: } \ell_i \sim \hat{\tau}_i}{\text{record }\{\overline{\mathsf{f}_i : \ell_i}\} \sim \{\overline{\mathsf{f}_i : \hat{\tau}_i}\}}\textsc{URecordMatch}$$

$$\frac{\text{for each } i\text{: } \ell_i \sim \hat{\tau}_i}{\text{variant }(s)\ \{\overline{\mathsf{A}_i\ (v_i) : \ell_i}\} \sim \langle\overline{\mathsf{A}_i\ \hat{\tau}_i}\rangle}\textsc{VariantMatch}$$

Wellformed layouts

$$\boxed{\ell \ \mathbf{wf}}$$

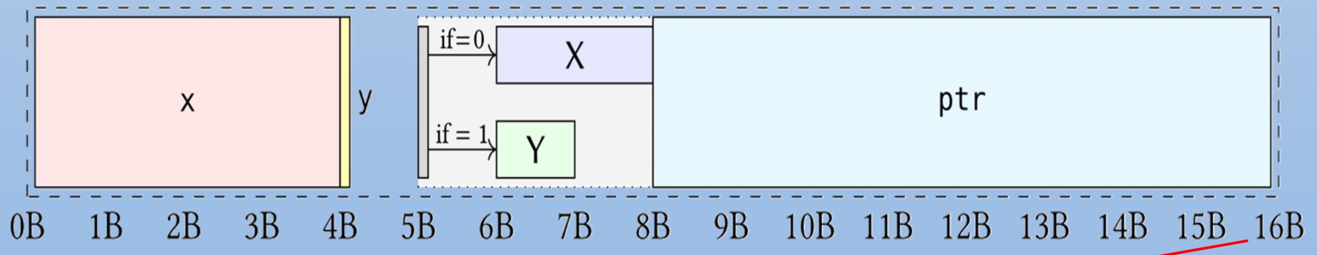
$$\frac{\text{for each } i\text{: } \ell_i \ \mathbf{wf} \qquad \text{for each } i \neq j\text{: taken}(\ell_i) \cap \text{taken}(\ell_j) = \emptyset}{\text{record } \{\overline{\mathsf{f}_i : \ell_i}\} \ \mathbf{wf}} \ \textsc{URecordWF}$$

$$\frac{\text{for each } i\text{: } \ell_i \ \mathbf{wf} \qquad v_i < 2^s \qquad \text{taken}(\ell_i) \cap \text{taken}(\mathsf{BitRange}\ (o, s)) = \emptyset \qquad \text{for each } i \neq j\text{: } v_i \neq v_j}{\mathsf{variant}\ (\mathsf{BitRange}\ (o, s))\ \{\overline{\mathsf{A}_i\ (v_i) : \ell_i}\} \ \mathbf{wf}} \ \textsc{VariantWF}$$

where taken$(\ell) \in \mathbb{N}$ returns the set of bit positions that ℓ occupies (defined recursively).

Compiling Cogent$^+$ to C

$\textbf{type}\ \texttt{Example} = \{$
$\quad \texttt{struct} : \#\{\texttt{x} : \texttt{U32}, \texttt{y} : \texttt{Bool}\},$
$\quad \texttt{ptr} : \{\ldots\},$
$\quad \texttt{sum} : \langle \texttt{X U16} \mid \texttt{Y U8} \rangle$
$\}$



Compilation

C

Type : array of 16 bytes

Getters/setters for each field

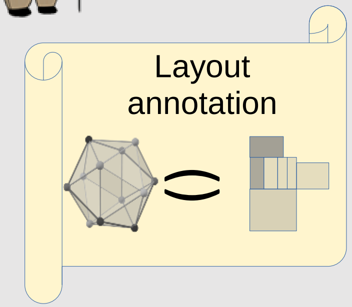
$get_{ptr} : byte[16] \rightarrow ptr$

$set_{ptr} : byte[16] \times ptr \rightarrow ()$

Example

A power controller system (libopencm3)

```
PWR_CR1.VOS = scale
```



Compilation

C

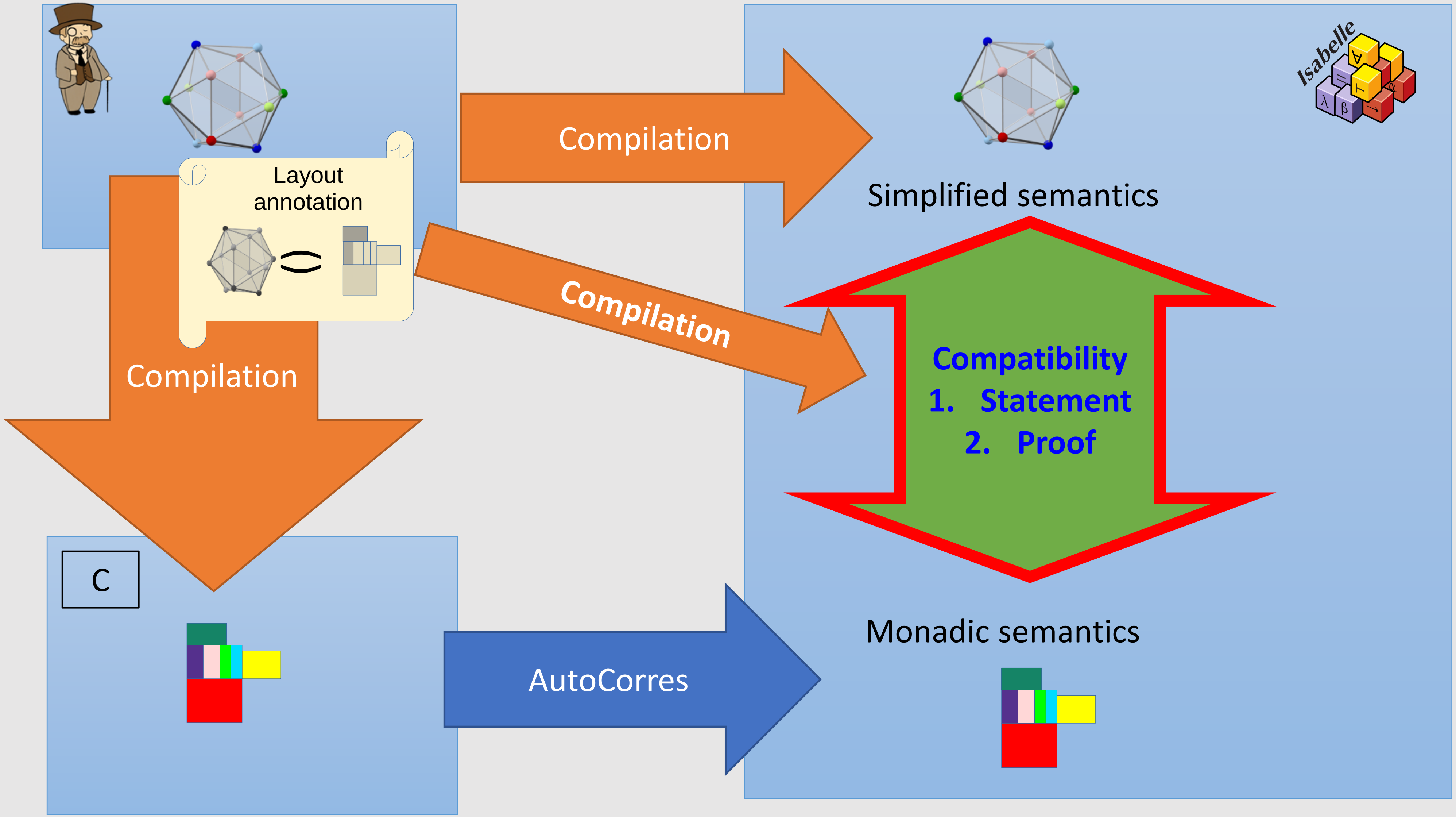
```
reg32 = PWR_CR1 & ~(PWR_CR1_VOS_MASK << PWR_CR1_VOS_SHIFT);
reg32 |= (scale & PWR_CR1_VOS_MASK) << PWR_CR1_VOS_SHIFT;
PWR_CR1 = reg32;
```

Unfolds as

```
set_VOS(PWR_CR1,scale);
```

Demo

Compiling Cogent+



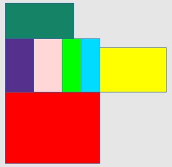
Stating compatibility

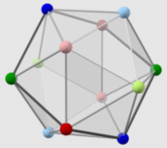
Monadic semantics	Simplified semantics
Type T_m	Type T_s
$f_m : A_m \to B_m$	$f_s : A_s \to B_s$

Compatibility between the two semantics
Relation $T_r \subset state \times T_m \times T_s$ **?**
(f_m, f_s) is compatible with (A_r, B_r)

- For an unannotated type, same T_r as before
- What about an annotated type?

T_m = byte[n]	$T_s = \{\dots, x : A, \dots\}$



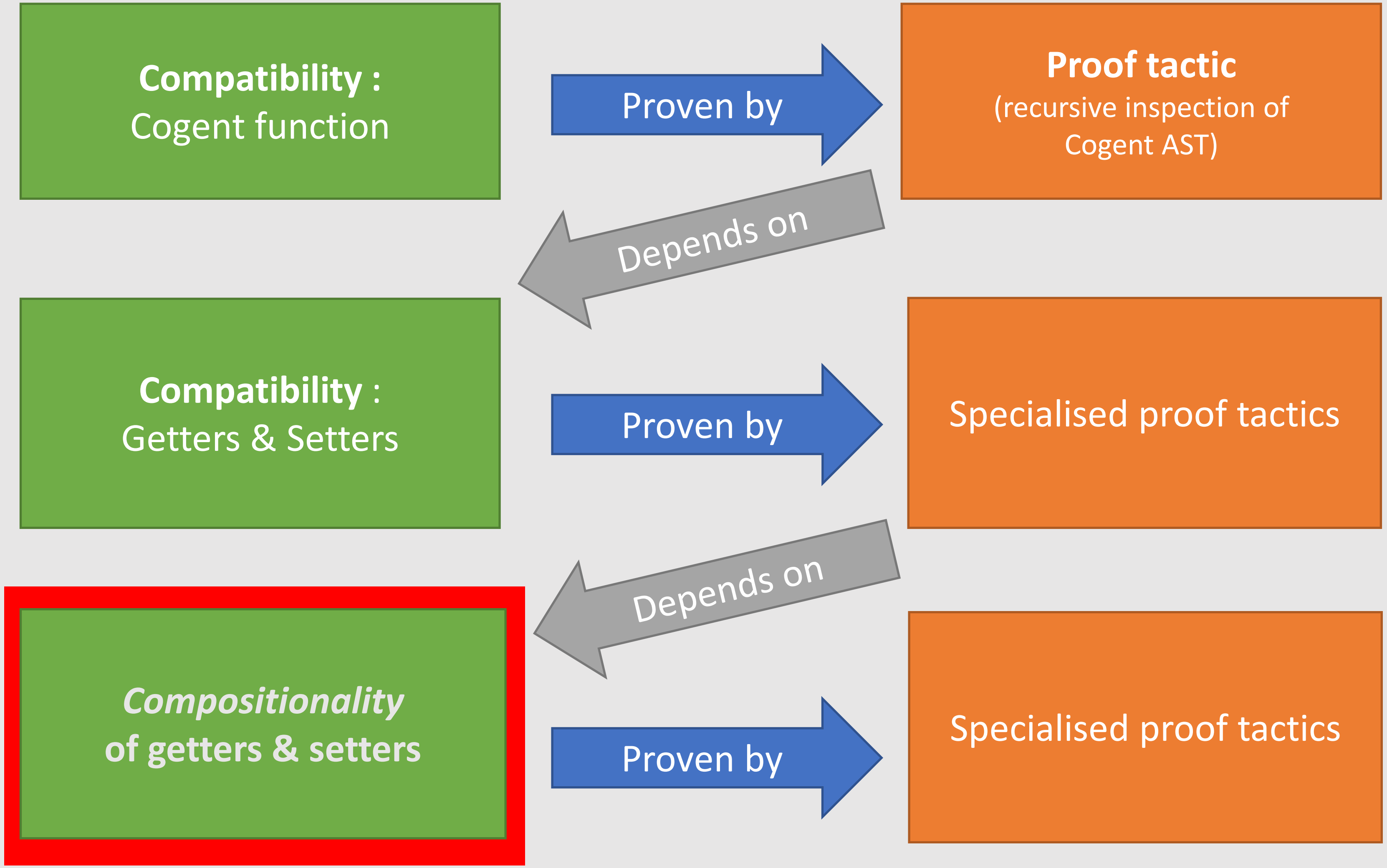


Decoding relation : « $T_r = \{(\dots, array, decode(array)\}$ »

Decodes the array according to the layout

$$decode : array \mapsto \left\{\begin{matrix} \dots \\ x := get_x(array) \\ \dots \end{matrix}\right\}$$

Proving compatibility



Compositionality of getters & setters

$$\text{«} \quad \begin{aligned} get_x(set_x(array, value)) &= value \\ get_x\left(set_y(array, value)\right) &= get_x(array) \end{aligned} \quad \text{»}$$

Enough to prove compatibility between the two semantics of getters & setters

Monadic semantics	Simplified semantics
T_m = byte[n]	$T_s = \{\dots, x : A, \dots\}$
get_x	$t \mapsto t.x$
set_x	$(t, a) \mapsto (t.x \coloneqq a)$

Towards an extensible Dargent

Dargent is limited.
For example, what if we want to target something else than a byte array?

Idea :
Instead of describing the layout, the user provides
- The target type (e.g., a byte array)
- An implementation of getters / setters
- The proofs of their compositionality properties

Enough to ensure compatibility!

Demo

Certifying getters & setters

Observation :
Compatibility between the semantics gives no guarantee that the generated getters / setters conform to the specified layout!

Additional generated formal guarantees

1. Generated setters do not flip any bit outside their layouts
2. Generated getters specialise a generic getter implemented in Isabelle (parameterised by a layout)

Demo

Some feature wishes

- Recursive types, Primitive arrays (Cogent)
- Controlling the layout of the field type in getters / setters

T= $\{\ldots, x : A, \ldots\}$

$get_x : byte[n] \rightarrow A$

$set_x : byte[n] \times A \rightarrow ()$

VS

$get_x : byte[n] \rightarrow byte[n_A]$

$set_x : byte[n] \times byte[n_A] \rightarrow ()$

Example : A = $< X\ U8 \mid Y\ U16 >$

$$get_x : byte[n] \rightarrow \begin{Bmatrix} tag : U8 \\ payload_X : U8 \\ payload_Y : U16 \end{Bmatrix}$$

VS

$$get_x : byte[n] \rightarrow byte[3]$$

Some feature wishes

- Simultaneous field updates

```
regs {
   timer_a_en = True,
   timer_a_input_clk =
       TIMEOUT_TIMEBASE_1_MS,
   timer_e_input_clk =
       TIMESTAMP_TIMEBASE_1_US }
```

Compiles to?

```
timer->regs->mux =
   TIMER_A_EN
  | (TIMEOUT_TIMEBASE_1_MS <<
     TIMER_A_INPUT_CLK)
  | (TIMESTAMP_TIMEBASE_1_US <<
     TIMER_E_INPUT_CLK) ;
```

A formally verified timer driver

Demo

Conclusion

Cogent + Dargent = First programming language with

1. Possible customisation of the layout of compiled types
2. Formal guarantees

Check our POPL ‘23 article:

Dargent: A Silver Bullet for Verified Data Layout Refinement