le cnam

Formal Methods for Critical Systems: A verified implementation of nested procedures*

Tristan Crolard [1]

ICAR'15

8-9 October 2015

Joint work with:

Maria-Virginia Aponte,[1] Pierre Courtieu,[1,2] Julia Lawall [3]

1. CNAM / Cedric / CPR team
2. INRIA / Gallium team
3. UPMC / LIP6 / Whisper team

* Research project funded by AdaCore, *the GNAT Pro Company*

Formal Methods for Critical Systems

Formal Methods for Critical Systems

based on a mathematical formalism

Formal Methods for Critical Systems

based on a mathematical formalism

life-critical or safety-critical

Formal Methods for Critical Systems

based on a mathematical formalism

life-critical or safety-critical

embedded systems

Formal Methods for Critical Systems

based on a mathematical formalism

life-critical or safety-critical

embedded systems

Formal methods are about:

- formal specifications
- mathematical proofs of properties

Formal Methods for Critical Systems

based on a mathematical formalism

life-critical or safety-critical

embedded systems

machine-checked

Formal methods are about:

- formal specifications
- mathematical proofs of properties

Machine-checked mathematical proofs

You might want to prove:

- some safety and security properties of your system
- the full correctness of your implementation with respect to its specification
- only the partial correctness of your implementation (no buffer overflow, for instance)

In any case, you need a formal specification of your system.

Machine-checked mathematical proofs

You might want to prove:

- some safety and security properties of your system
- the full correctness of your implementation with respect to its specification
- only the partial correctness of your implementation (no buffer overflow, for instance)

In any case, you need a formal specification of your system.

Of course, testing is still allowed and a formal specification is also required in this case (when mixing tests and proofs).

Formal Methods: logics and tools

expressive

Higher-order logics

full correctness

First-order logics

partial correctness

decidable

Specialized logics

specific properties

Formal Methods: logics and tools

expressive | *interactive*

Higher-order logics ⟷ Proof assistants

full correctness

First-order logics ⟷ Provers and solvers

partial correctness

Specialized logics ⟷ Model checkers

specific properties

decidable | *automatic*

Formal Methods: logics and tools

expressive

interactive

Higher-order logics ⟷ Proof assistants

full correctness

Program logics

First-order logics ⟷ Provers and solvers

partial correctness

decidable

automatic

Specialized logics ⟷ Model checkers

specific properties

Limits of formal methods

“The correspondence between our formal models of programs and the actual behavior of real systems is limited by three factors:

- the behavior of the programming language,
- the operating system,
- and the underlying hardware.

For safety-critical systems, these limitations are crucially important and we cannot assume that a program is correct just because it has been proved.”

Seven Myths of Formal Methods
Anthony Hall, Praxis Sytems, September 1990

Two success stories about formal methods

- The seL4 project developed at NICTA (SSRG).
  - seL4 is a formally-verified microkernel
  - Developed since 2006.
  - First public release in 2011 (open source since 2014).
- The CompCert project developed at INRIA (Gallium team).
  - CompCert is a formally-verified C compiler
  - Developed since 2004.
  - First public release in 2008.

The seL4 project

seL4 is a high-performance general-purpose microkernel, that provides address spaces, threads, IPC and authorisation capabilities

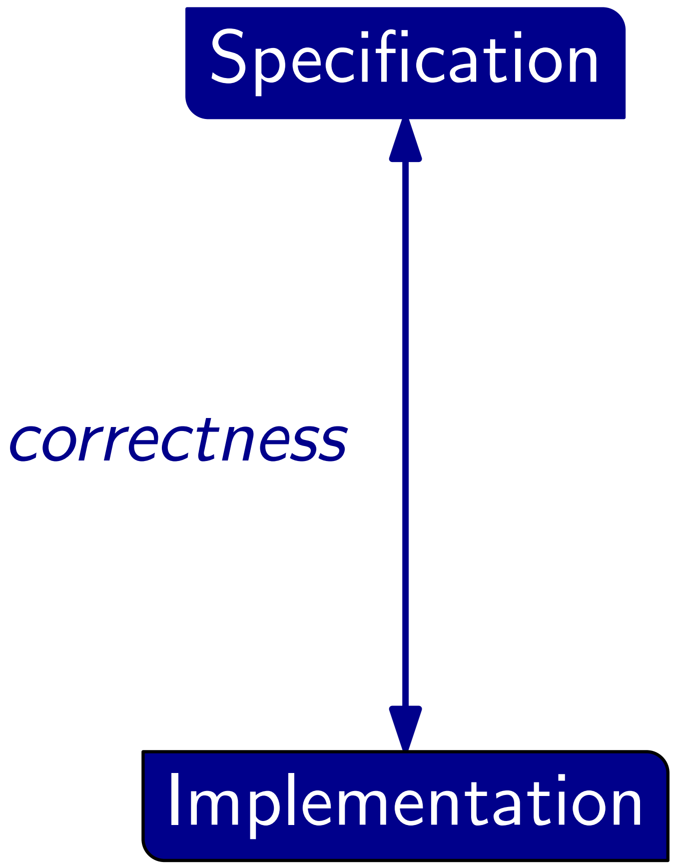
- Formal proof of correctness down to binary level
- Developed for ARM and Intel processors
- The fastest existing microkernel (faster than L4)
- 10,000 lines of code
- 200,000 lines of proof
- about 30 person.years

The CompCert project

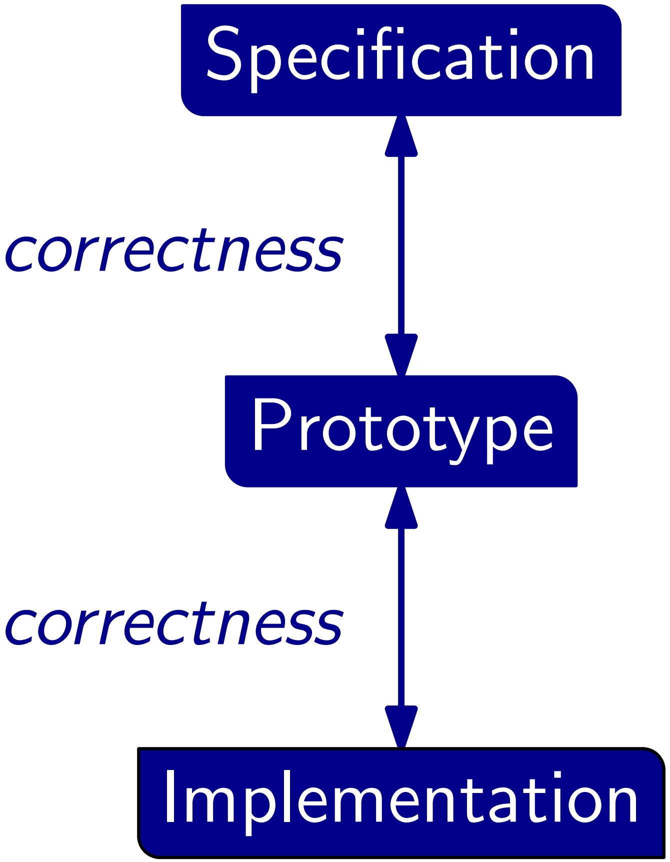
A formally-verified optimizing standard C compiler

- Formal proof of correctness down to binary level
- Developed for PowerPC, ARM and Intel processors
- Generated code only 20% slower than gcc -O2
- 15,000 lines of code
- 100,000 lines of proof
- about 6 person.years

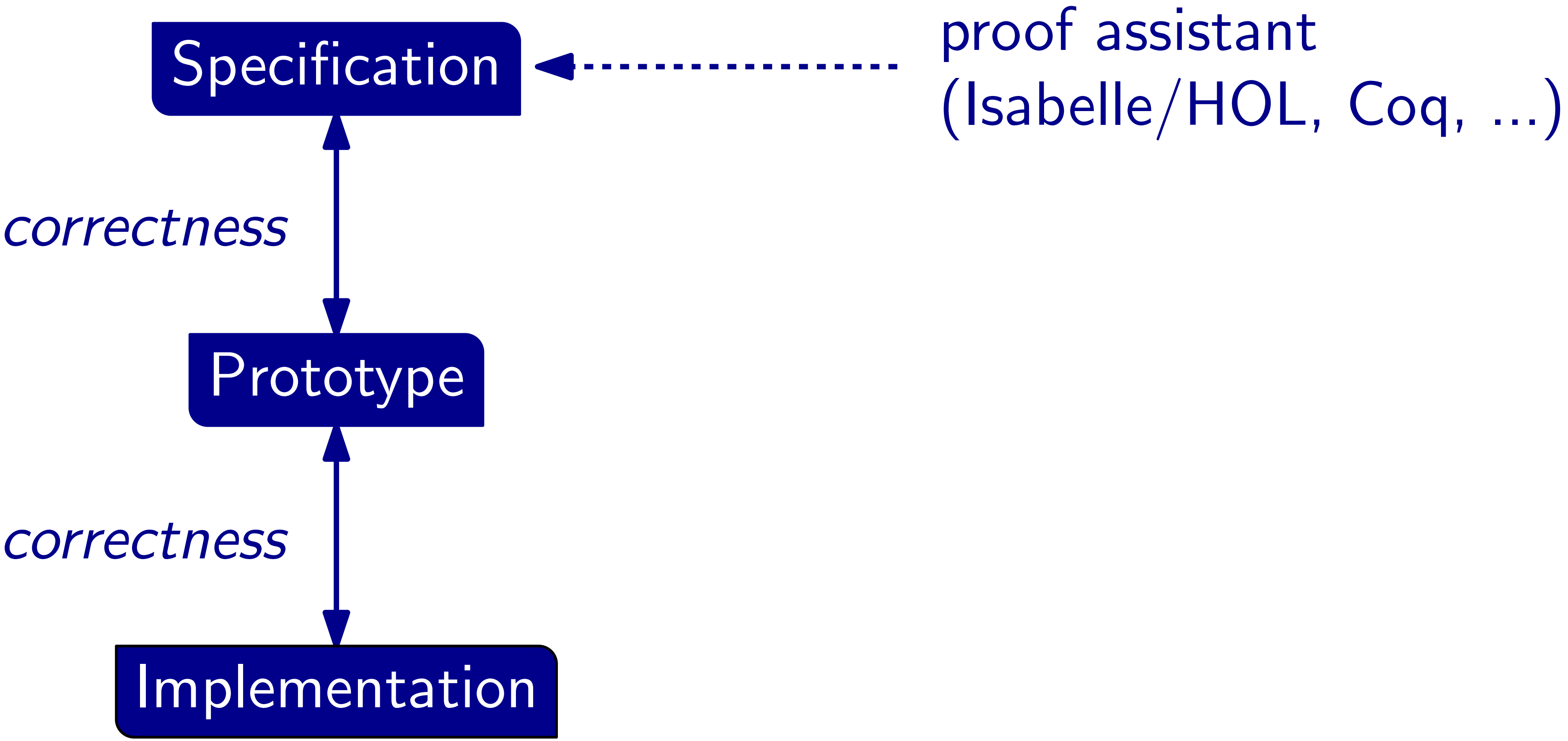
Proof Architecture



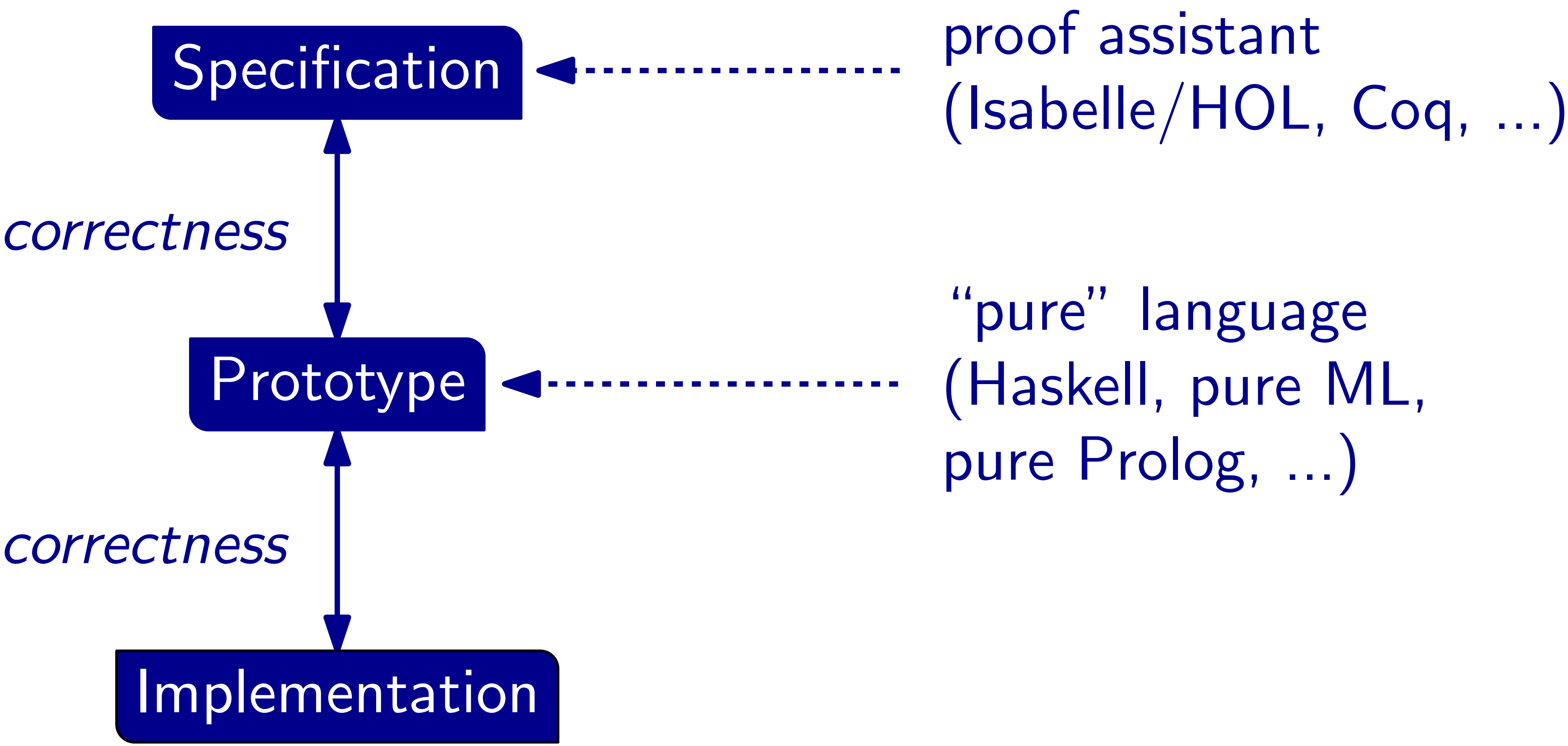
Proof Architecture



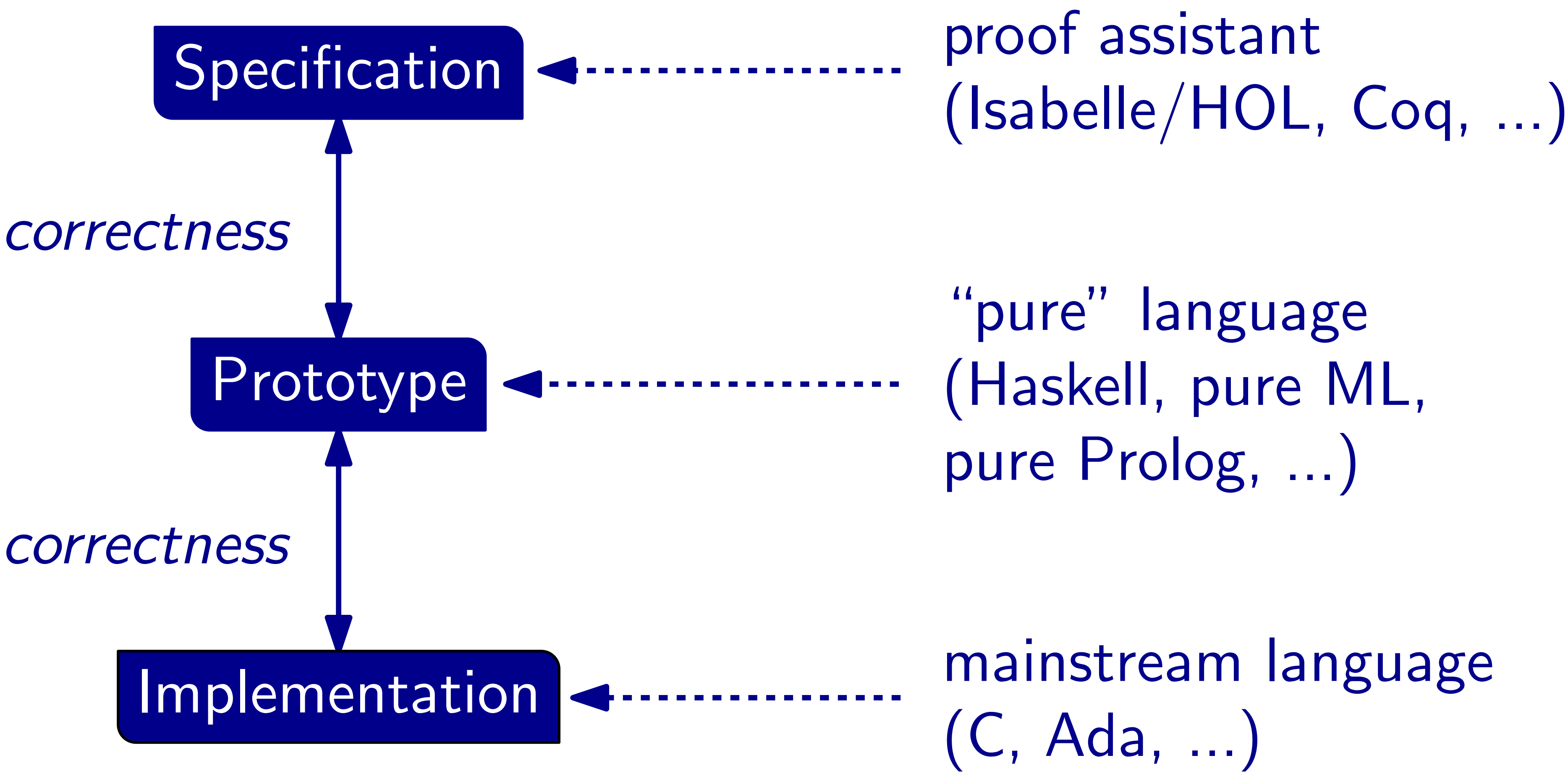
Proof Architecture



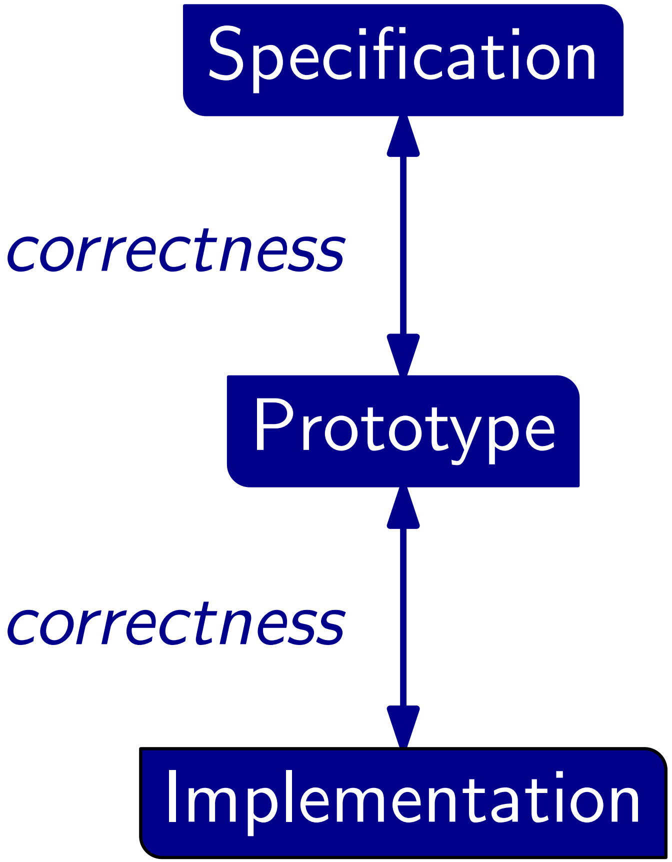
Proof Architecture



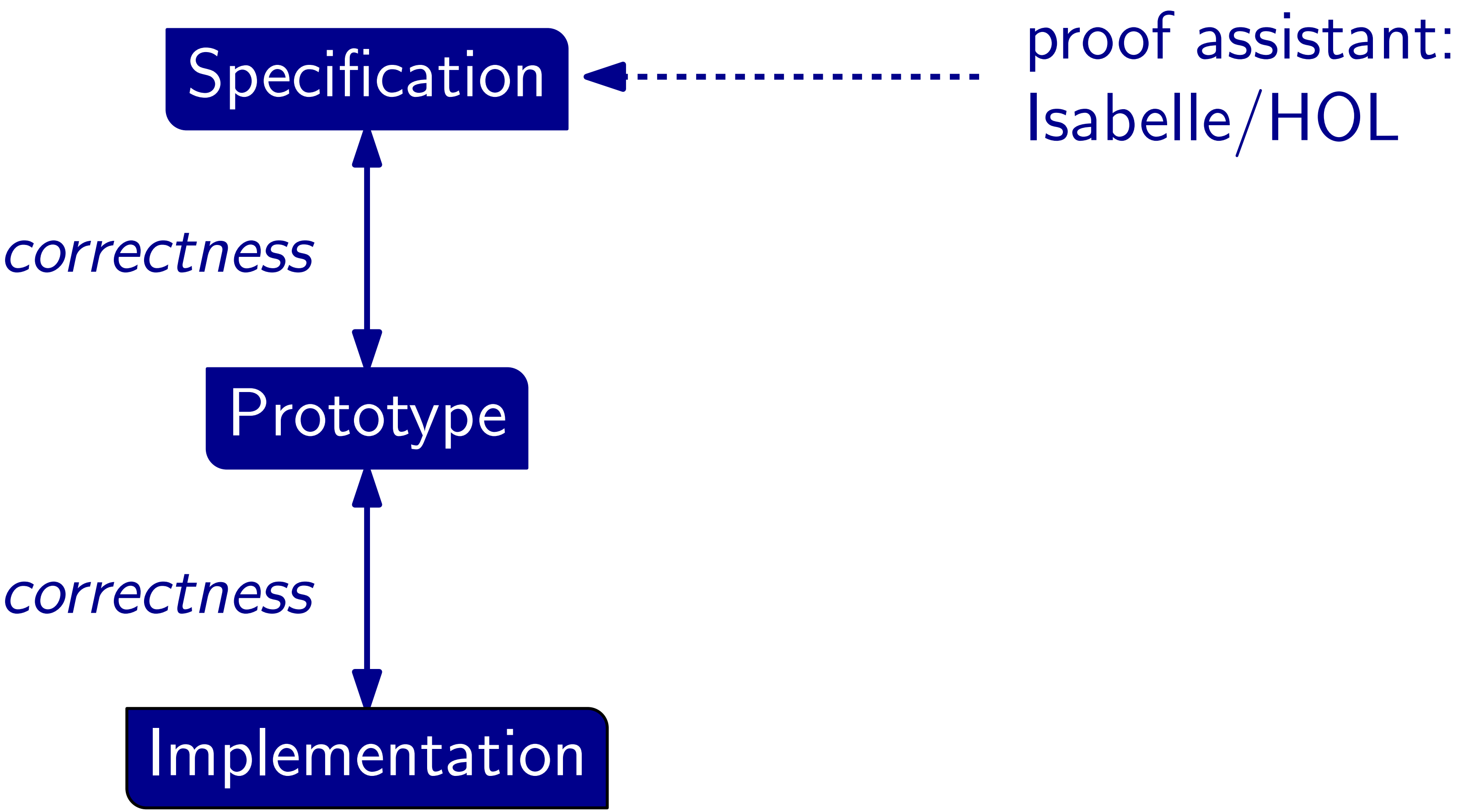
Proof Architecture



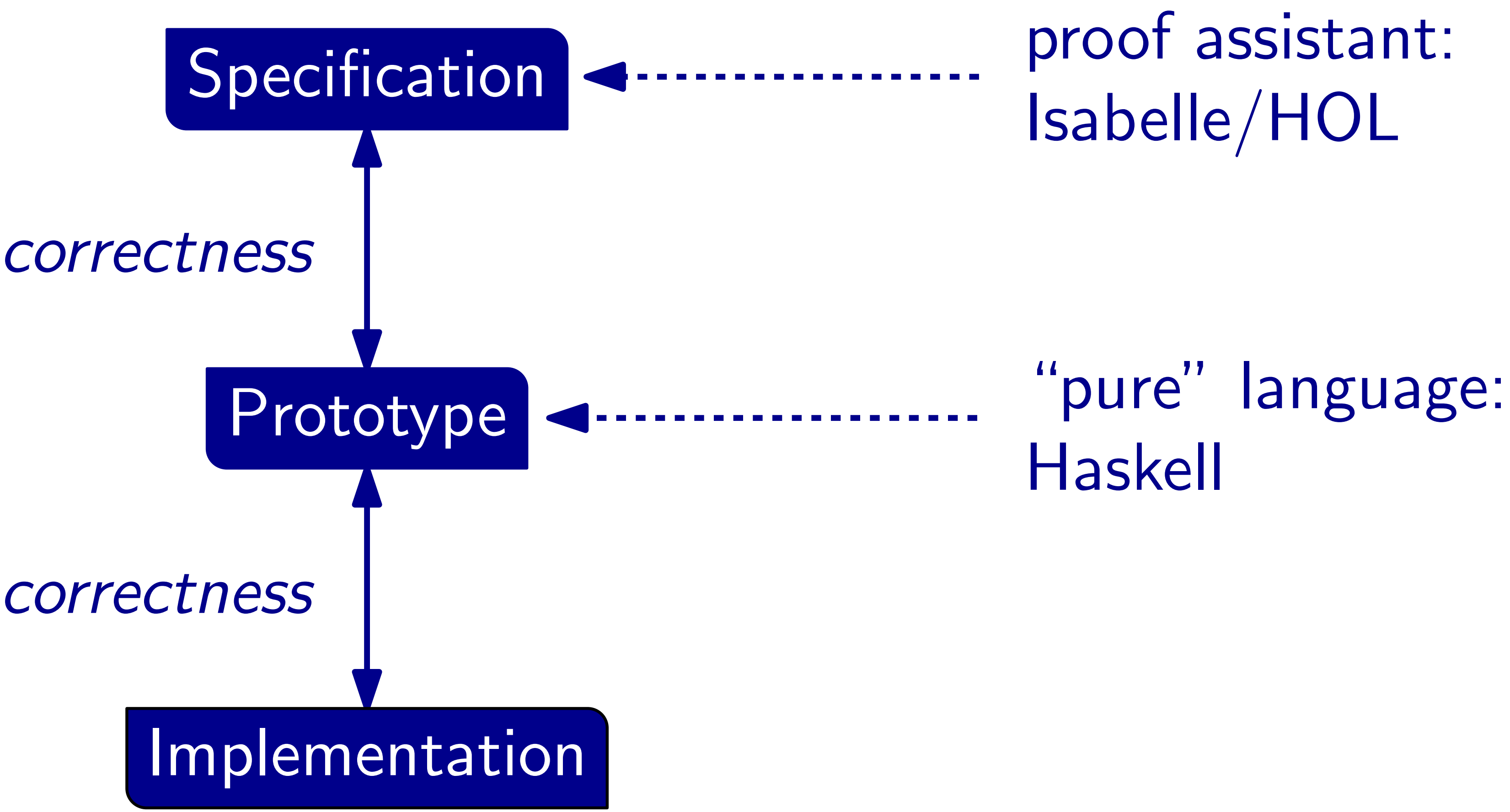
Proof Architecture: seL4



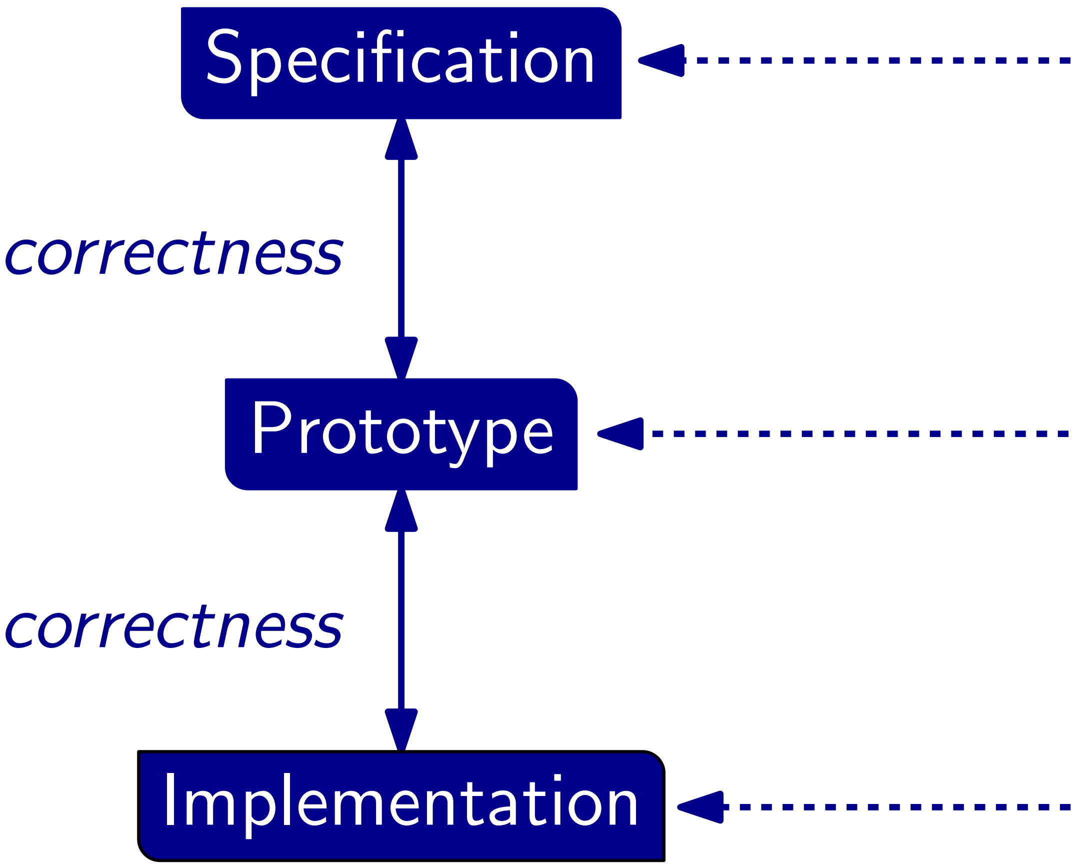
Proof Architecture: seL4



Proof Architecture: seL4



Proof Architecture: seL4

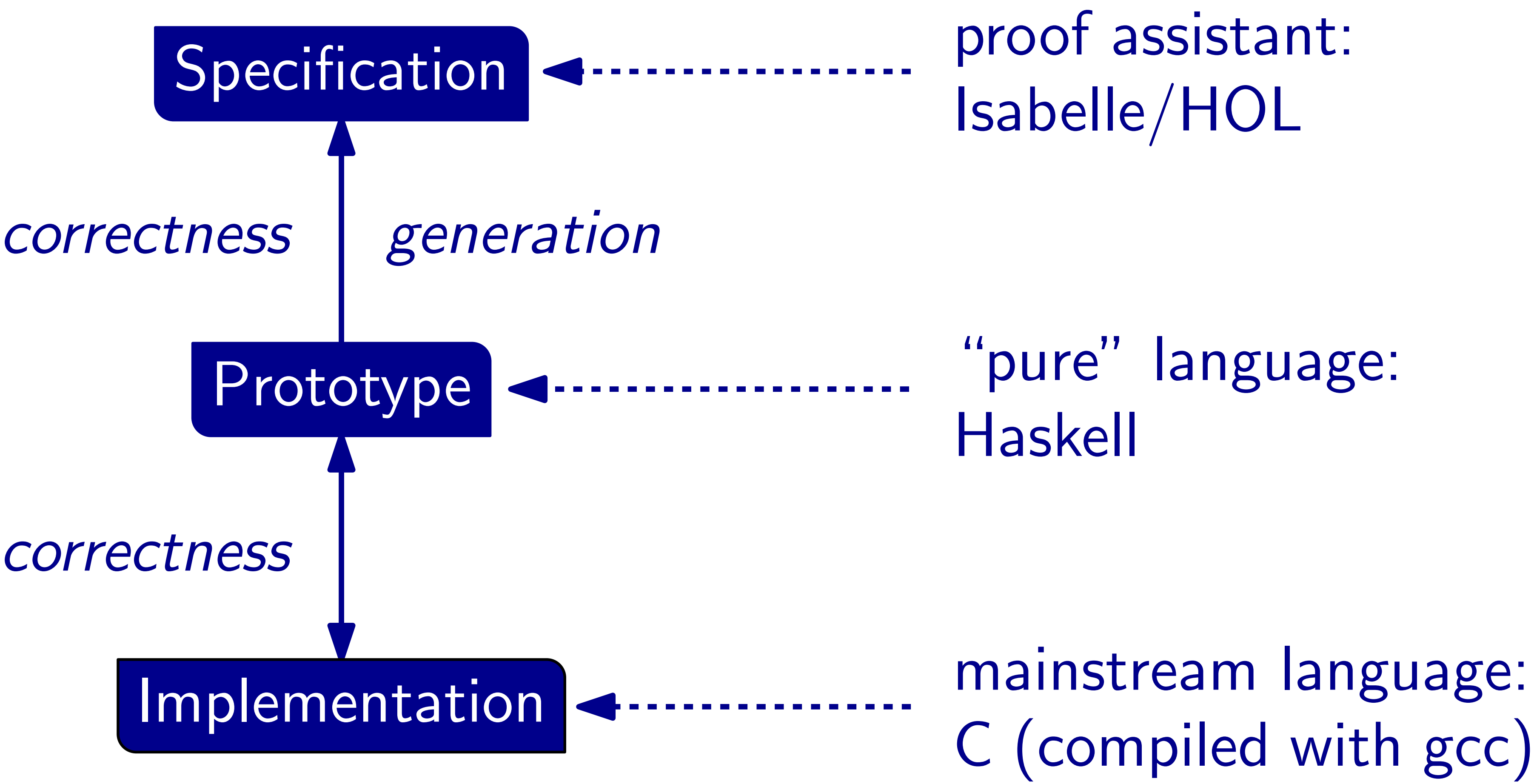


proof assistant:
Isabelle/HOL

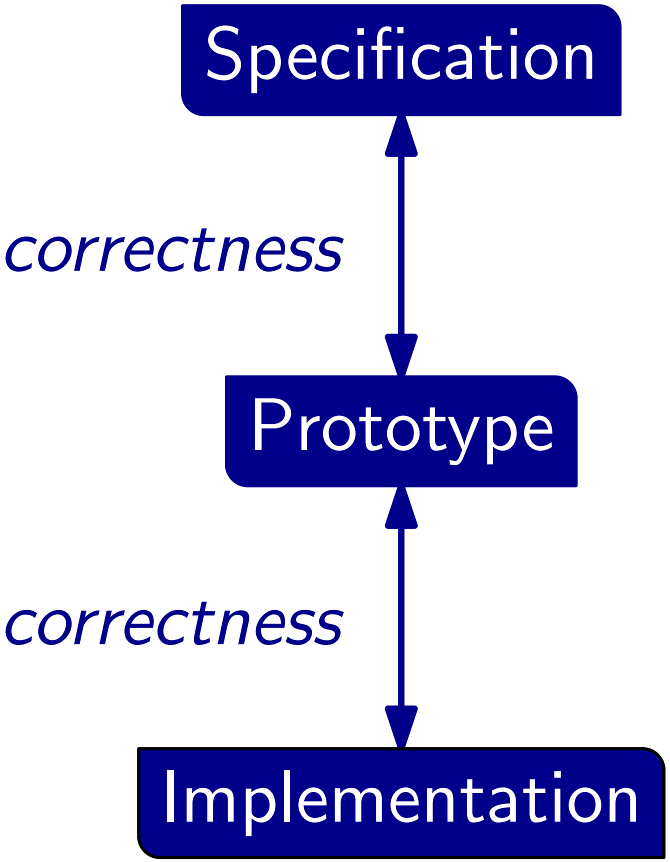
“pure” language:
Haskell

mainstream language:
C (compiled with gcc)

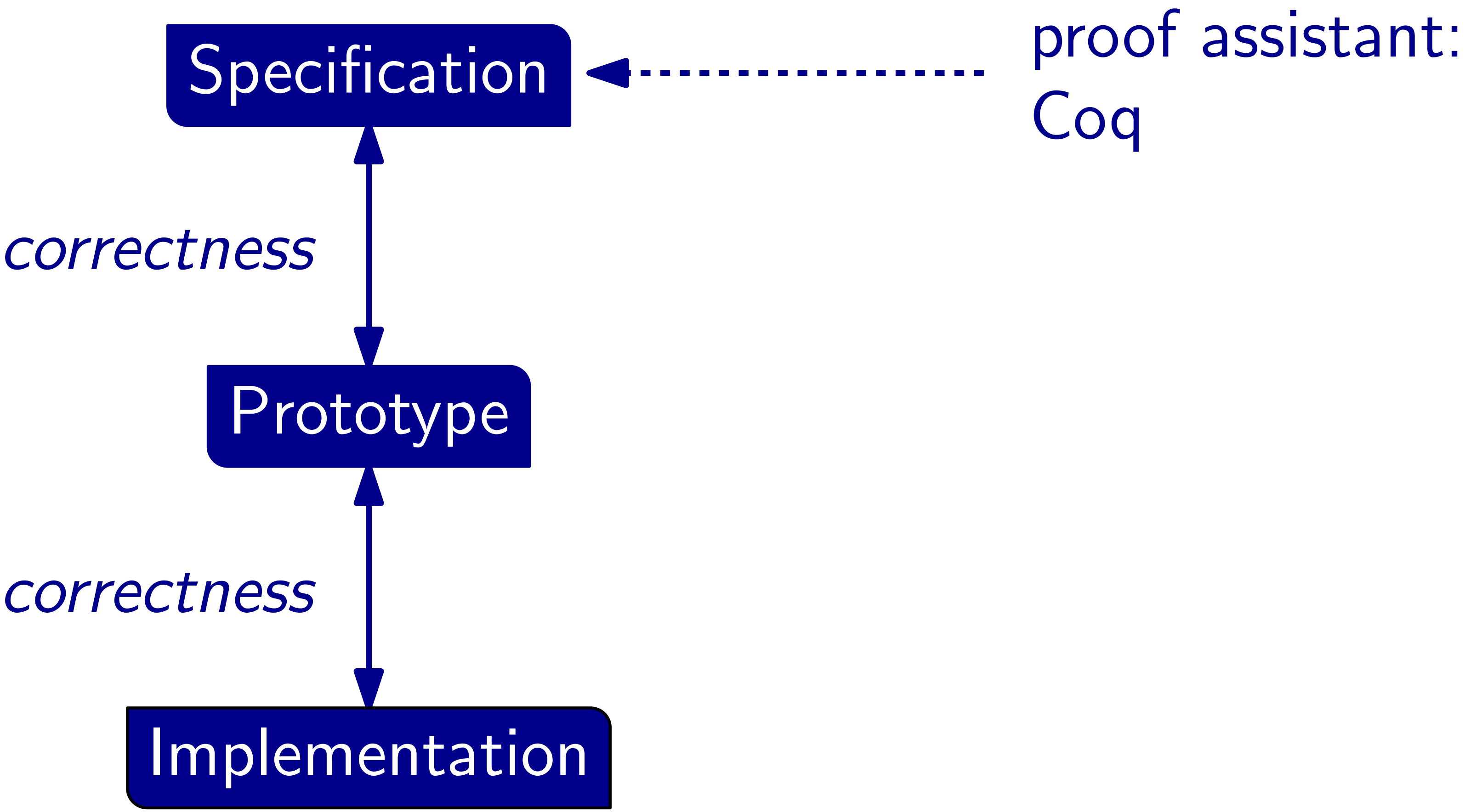
Proof Architecture: seL4



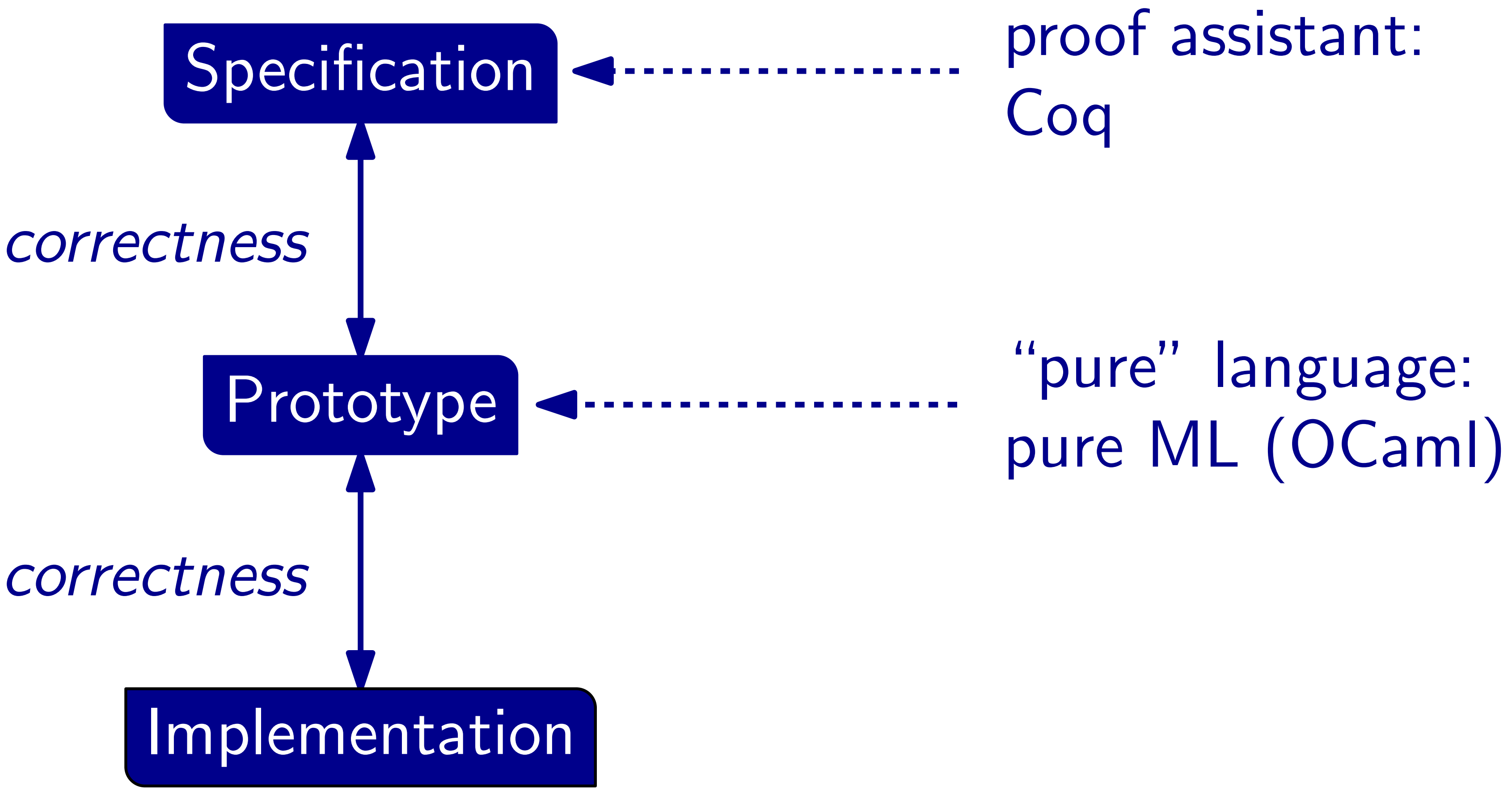
Proof Architecture: CompCert



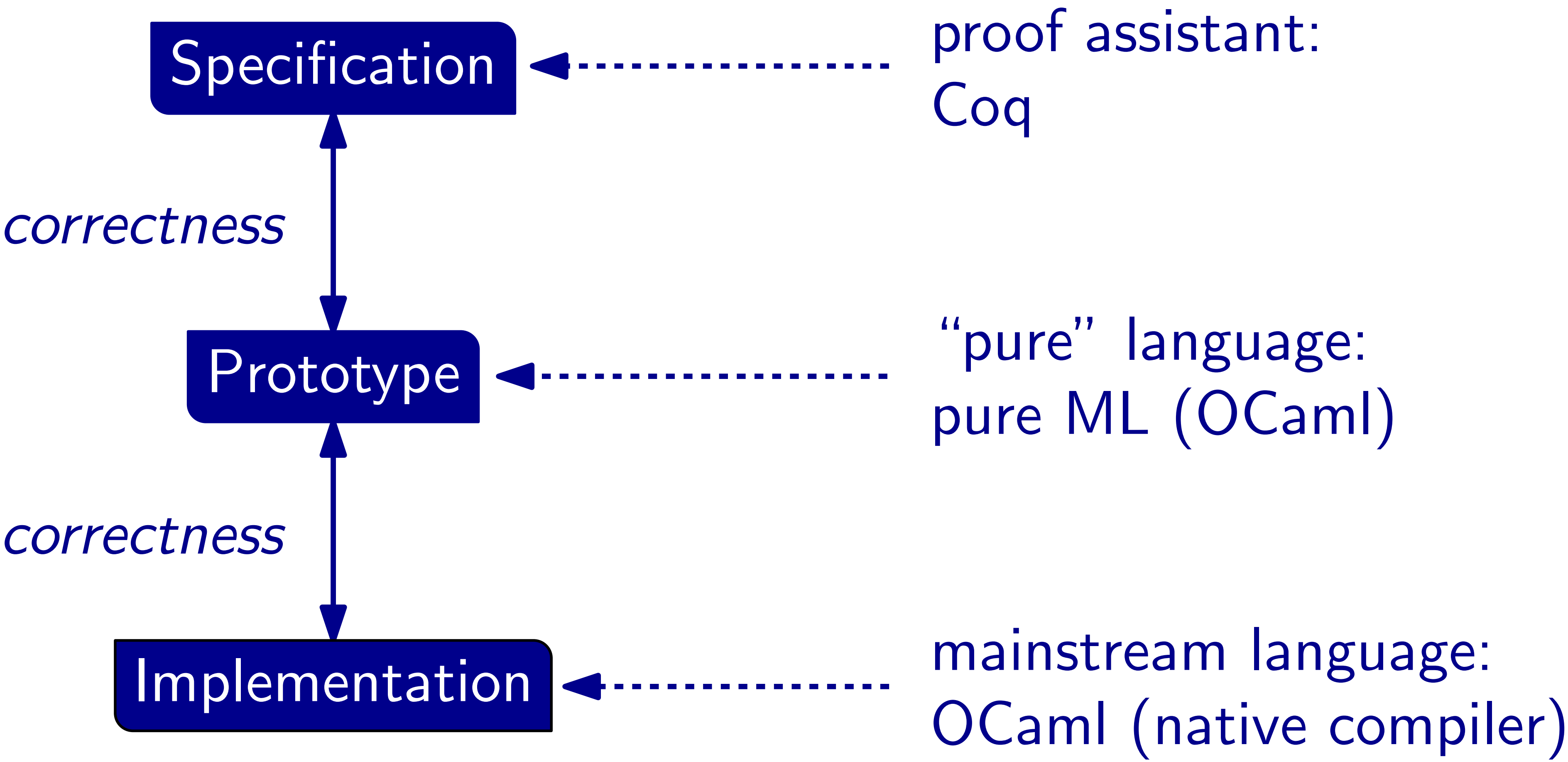
Proof Architecture: CompCert



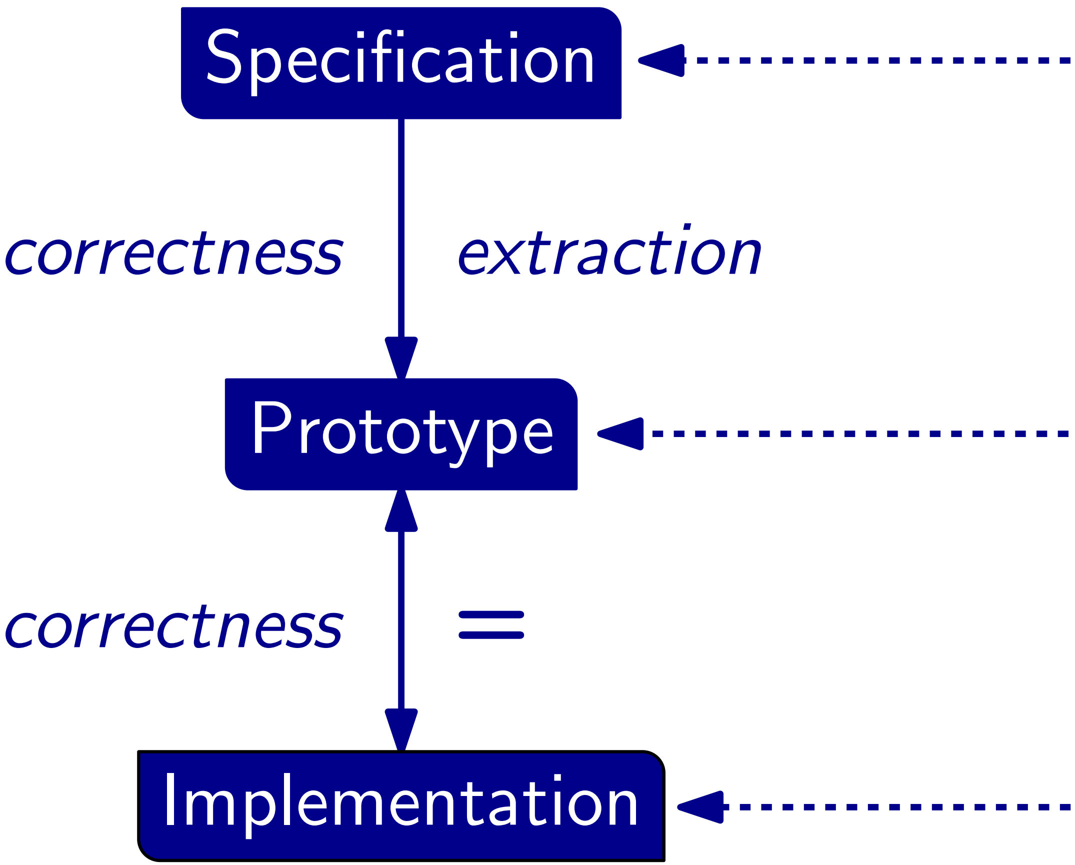
Proof Architecture: CompCert



Proof Architecture: CompCert



Proof Architecture: CompCert



proof assistant:
Coq

"pure" language:
pure ML (OCaml)

mainstream language:
OCaml (native compiler)

How to prove the correctness of a compiler ?

- A compiler translates a source program into a target program
- The translation is correct if the target program has the same behaviour as the source program
- Formally, we need some mathematical abstraction of the behaviour (a semantics)

How to prove the correctness of a compiler ?

- Let us write $p \sim p'$ when p and p' have the same behaviour
- Let us call * the translation performed by the compiler

$$\boxed{\text{source program}} \qquad\qquad \boxed{\text{target program}}$$

$$p \xrightarrow{\quad\quad\star\quad\quad} p^\star$$

- Correctness: For *any* source program p,

$$p \sim p^\star$$

How to prove the correctness of a compiler ?

There are two options:

- For each program p, prove $p \sim p^\star$
  - *translation validation* approach [Pnuelli 1998]
  - first-order formulas (mostly automatic)
  - works for a regular compiler (for instance gcc)
  - successfully used in the seL4 project
- Prove $\forall p, p \sim p^\star$
  - higher-order formula (requires a proof-assistant)
  - successfully used in the CompCert project

The CompCert project: a verified C compiler

"The CompCert project investigates the formal verification of realistic compilers usable for critical embedded software.

- Such verified compilers come with a mathematical, machine-checked proof that the generated executable code behaves exactly as prescribed by the semantics of the source program.
- By ruling out the possibility of compiler-introduced bugs, verified compilers strengthen the guarantees that can be obtained by applying formal methods to source programs."

The CompCert project
Xavier Leroy

Can you trust your C compiler?

"We created a tool that generates random C programs, and [...] every compiler that we tested has been found to crash and also to silently generate wrong code when presented with valid inputs."

Finding and understanding bugs in C compilers.
Yang et al. University of Utah, PLDI 2011.

Can you trust your C compiler?

"We created a tool that generates random C programs, and [...] every compiler that we tested has been found to crash and also to silently generate wrong code when presented with valid inputs."

"The striking thing about our CompCert results is that the middleend bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors."

Finding and understanding bugs in C compilers.
Yang et al. University of Utah, PLDI 2011.

Critical Systems: programming languages

- Embedded systems are usually developed in C or Ada (with some assembly code)
- Critical systems are developed in subsets of these languages, such as MISTRA C or SPARK Ada.
- Dedicated frameworks also generate either C or (SPARK) Ada source code.
  - B Method
  - SCADE Suite
  - Simulink

Ada: a language designed for embedded systems

- First standardized version in 1983
- Ada is an algol-like language:
  - strong static typing
  - real procedures with proper parameter modes
  - packages (modules)
  - generics
  - support for concurrency
  - support for real-time systems
  - object-oriented (since 1995)
  - support for contracts (since 2012)

Who is using Ada?

Ada is often used in large critical systems:

- Commercial Aviation:
  Most Airbus and Boeing air-planes
- Commercial Rockets:
  Ariane 4 and 5
- Railway Transportation:
  Paris drive-less Metro line 14
- ...

SPARK: a strict subset of Ada

- Developed by ALTRAN Praxis and AdaCore
- Supported by any standard Ada 2012 compiler
- Well-defined subset of Ada designed for Critical Systems
  - ~~Pointers~~
  - ~~Effects in expressions~~
  - ~~Parameter-induced aliasing~~
  - ~~Exception handler~~
- Static analysis (SPARK tools)
  - to ensure that contracts are met (pre/post conditions)
  - to ensure that runtime checks never fail

SPARK: a strict subset of Ada

- Developed by ALTRAN Praxis and AdaCore
- Supported by any standard Ada 2012 compiler
- Well-defined subset of Ada designed for Critical Systems
  - ~~Pointers~~
  - ~~Effects in expressions~~
  - ~~Parameter-induced aliasing~~
  - ~~Exception handler~~
- Static analysis (SPARK tools)
  - to ensure that contracts are met (pre/post conditions)
  - to ensure that runtime checks never fail

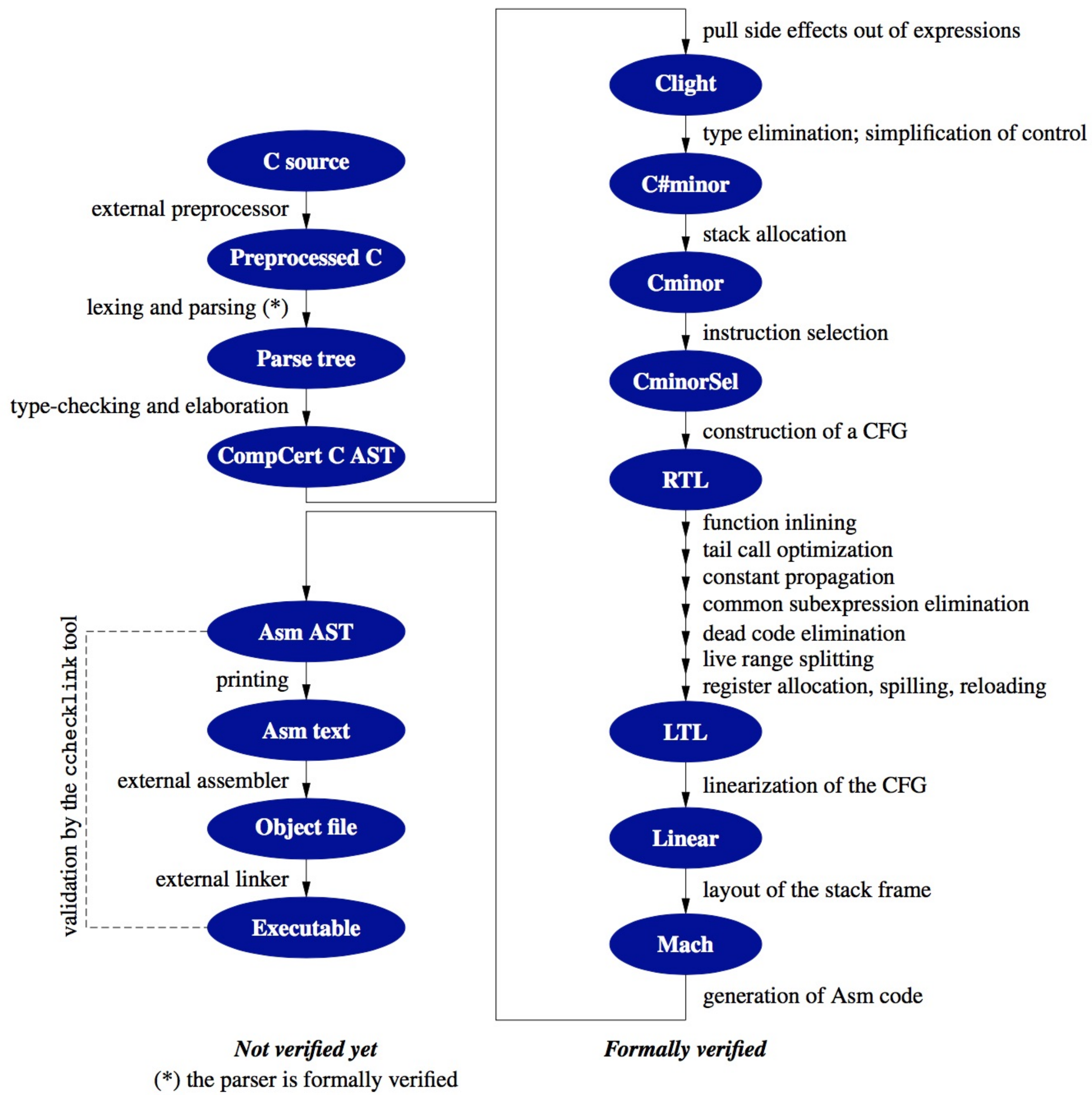
What about a verified SPARK Ada Compiler?

- A large on-going project, in collaboration with AdaCore and SAnToS Lab (Kansas State University), since 2012.
- Current state of the formal specification:
  - small fragment of Ada (similar to C in expressiveness)
  - some runtime checks (overflows)
  - nested procedures
- Unsupported features:
  - packages
  - generics
  - contracts
  - ...

What about a verified SPARK Ada Compiler?

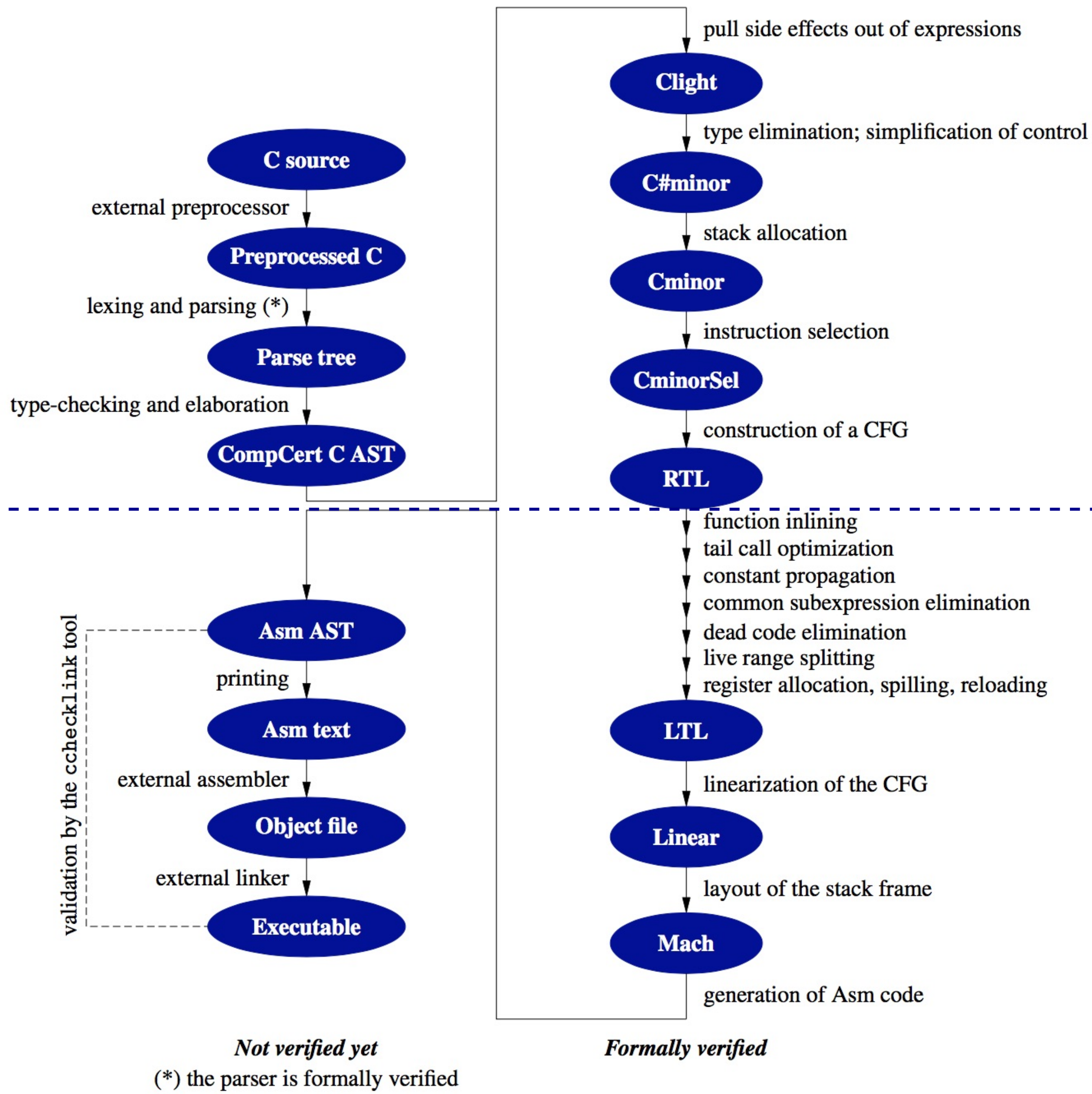
- Current state of the compiler:
  - a SPARK Ada frontend to CompCert
  - lexer and parser from *gnat* (developed by AdaCore)
  - converter to Coq AST (developed by SAnToS Lab)
  - proof-of-concept compiler (developed by P. Courtieu)
  - nested procedures (work in progress)
- Current state of the proofs:
  - correctness of the compiler (P. Courtieu)
  - absence of runtime error (P. Courtieu and SAnToS Lab)
  - nested procedures (this talk)

Architecture of the CompCert C compiler



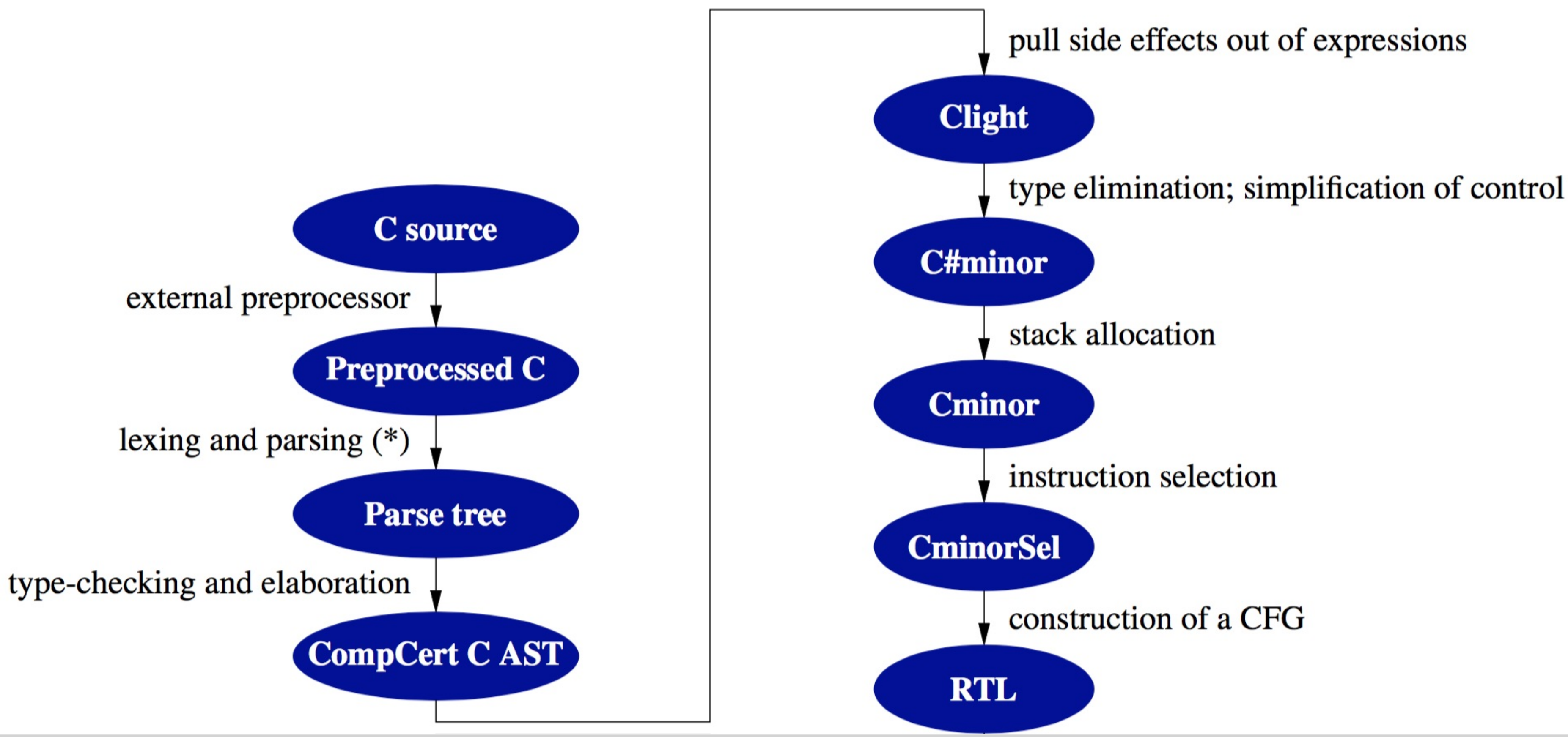
Architecture of the CompCert C compiler

Front-end



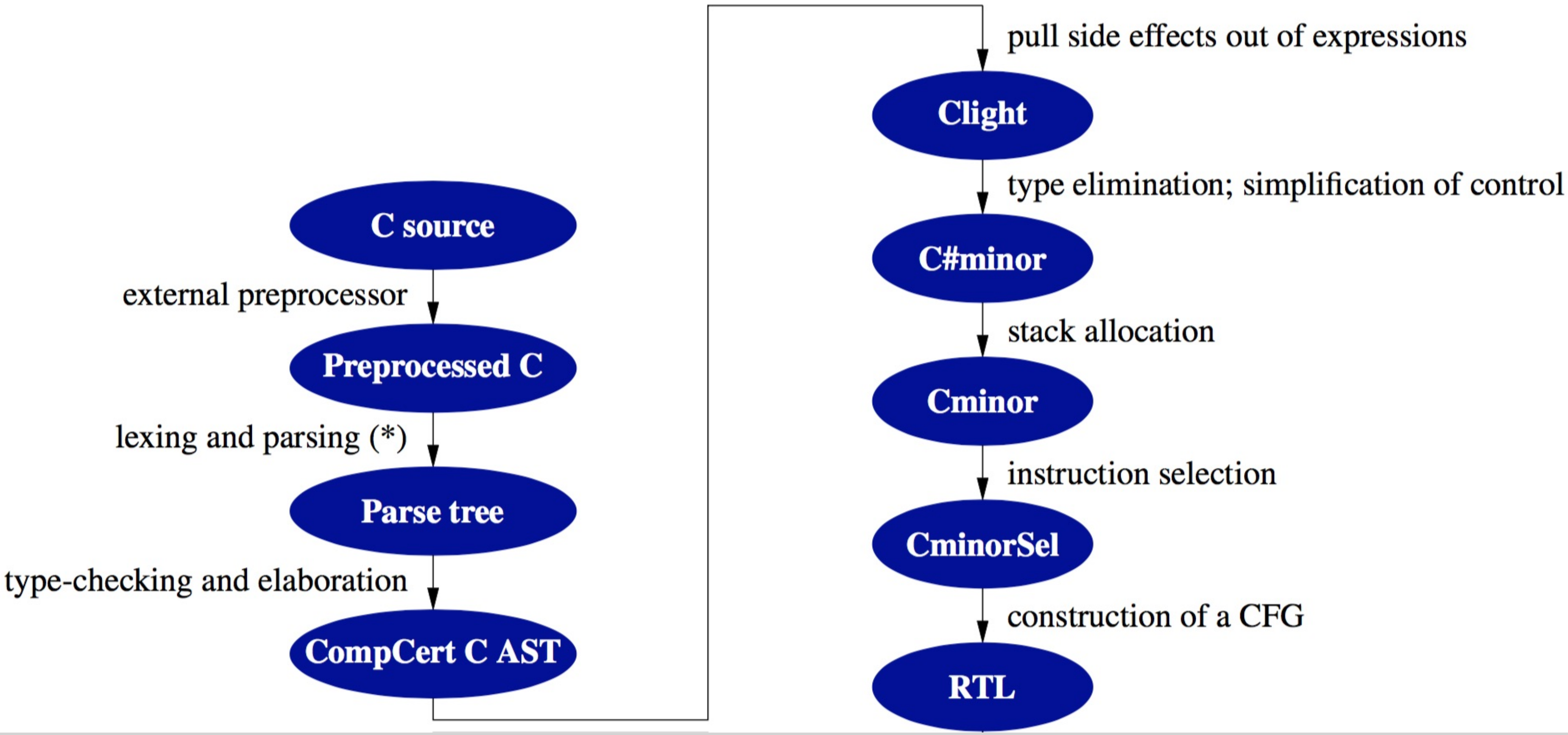
Back-end

Architecture of the CompCert C compiler



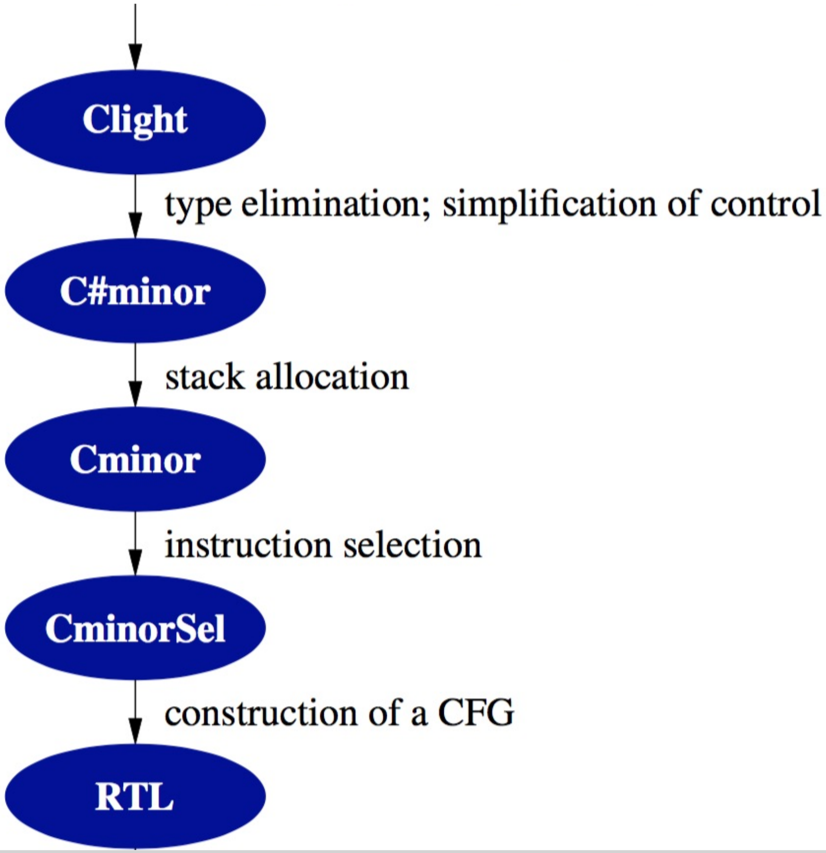
Architecture of the CompCert C compiler

- lexer and parser are specific to the C language



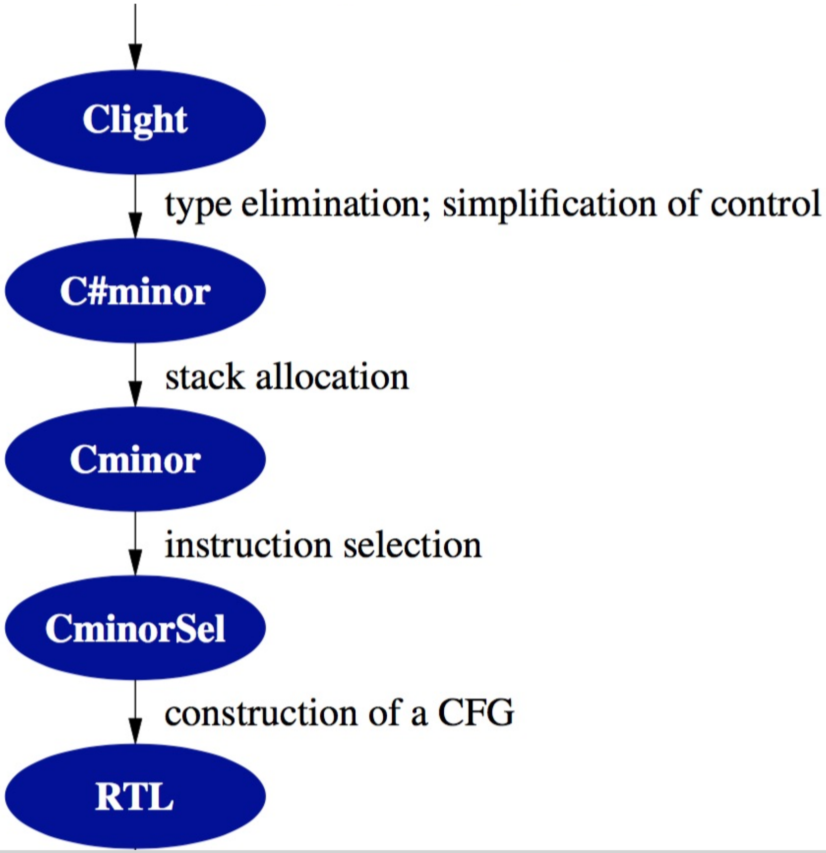
Architecture of the CompCert C compiler

- lexer and parser are specific to the C language



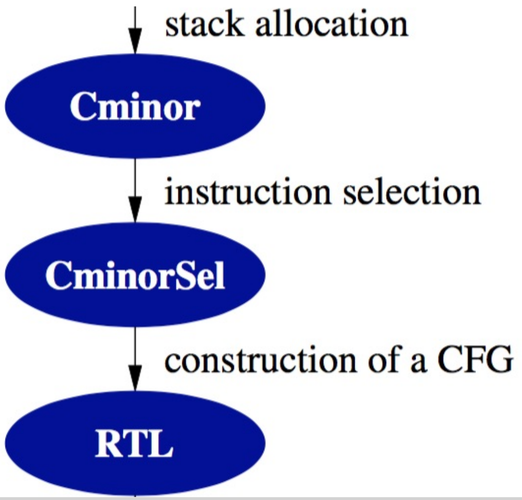
Architecture of the CompCert C compiler

- lexer and parser are specific to the C language
- Clight and C#minor are still too close to the C language



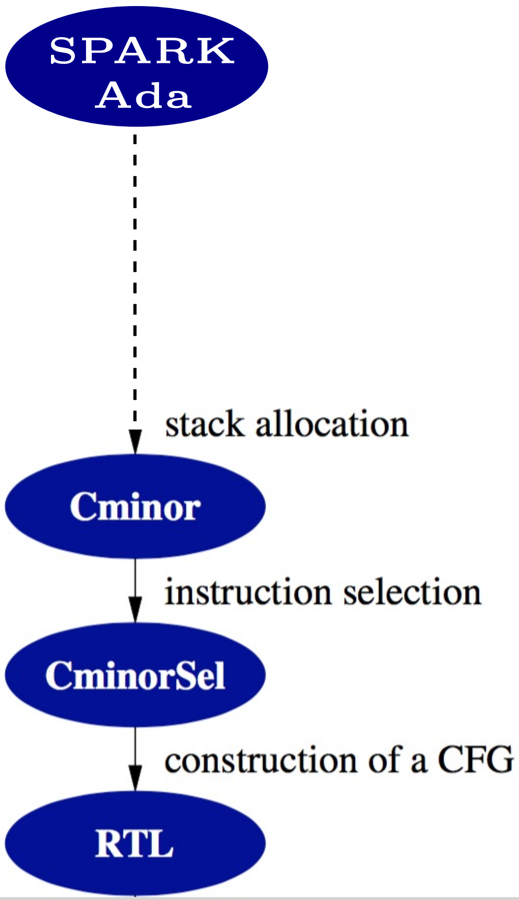
Architecture of the CompCert C compiler

- lexer and parser are specific to the C language
- Clight and C#minor are still too close to the C language



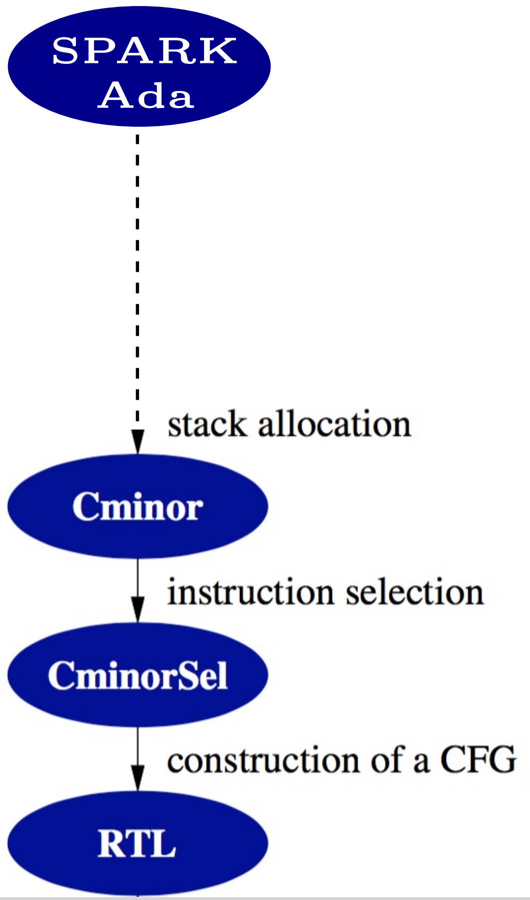
Architecture of the CompCert C compiler

- lexer and parser are specific to the C language
- Clight and C#minor are still too close to the C language



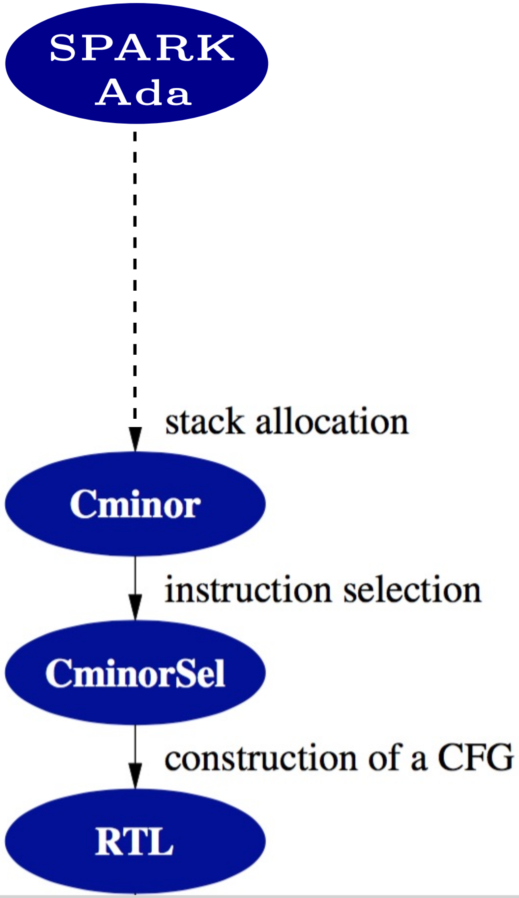
Architecture of the CompCert C compiler

- lexer and parser are specific to the C language
- Clight and C#minor are still too close to the C language
- SPARK Ada is much larger than the C language:
  - nested procedures
  - packages
  - generics
  - contracts
  - ...



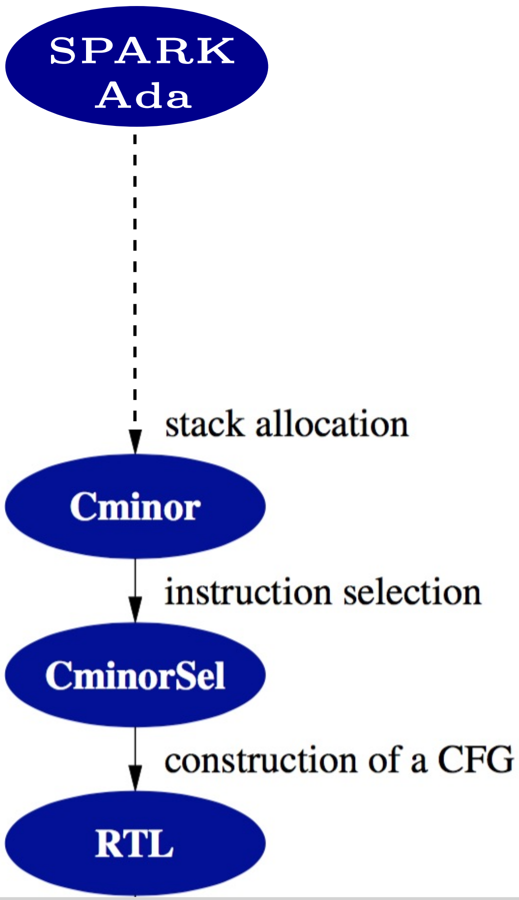
Architecture of the CompCert C compiler

- lexer and parser are specific to the C language
- Clight and C#minor are still too close to the C language
- SPARK Ada is much larger than the C language:
  - nested procedures
  - packages
  - generics
  - contracts
  - ...
- should require several intermediate languages



Architecture of the CompCert C compiler

- lexer and parser are specific to the C language
- Clight and C#minor are still too close to the C language
- SPARK Ada is much larger than the C language:
  - nested procedures
  - packages
  - generics
  - contracts
  - ...
- should require several intermediate languages



Ada 2012 source example


```
package Sorting
   with SPARK_Mode
is
   subtype Index is  Integer  range 1..100;
   type Vector is  array  (Index) of Integer ;

   procedure Swap(I, J :  Index;  V: in  out Vector)
     with Post => V = V'Old'Update (I => V'Old (J), J => V'Old (I));

   procedure Sort(V :  in  out Vector)
     with Post => (for all  X in  V'First + 1 ..  V'Last => (V(X - 1) <= V(X)));

end Sorting;
```

Implementing Swap as a **global** procedure


```
package body Sorting
   with SPARK_Mode
is

   procedure Swap(I, J : Index; V: in out Vector) is
       Aux: Integer ;
   begin
       Aux := V(I);
       V(I) := V(J);
       V(J) := Aux;
   end Swap;

   procedure Sort(V : in out Vector) is
   begin
      -- some code using Swap
   end Sort;

end Sorting;
```

Implementing Swap as a **nested** procedure

```
package body Sorting
   with SPARK_Mode
is

   procedure Sort(V : in out Vector) is

      procedure Swap(I, J : Index; V: in out Vector) is
          Aux: Integer ;
      begin
          Aux := V(I);
          V(I) := V(J);
          V(J) := Aux;
      end Swap;

   begin
      -- some code using Swap
   end Sort;

end Sorting;
```

Implementing Swap as a **nested** procedure


```
package body Sorting
   with SPARK_Mode
is

   procedure Sort(V : in out Vector) is

      procedure Swap(I, J : Index; V: in out Vector) is
         Aux: Integer ;
      begin
         Aux := V(I);
         V(I) := V(J);
         V(J) := Aux;
      end Swap;

   begin
      -- some code using Swap
   end Sort;

end Sorting;
```

no longer required

Implementing Swap as a **nested** procedure


```
package body Sorting
   with SPARK_Mode
is

   procedure Sort(V : in out Vector) is

      procedure Swap(I, J : Index) is
          Aux: Integer ;
      begin
          Aux := V(I);
          V(I) := V(J);
          V(J) := Aux;
      end Swap;

   begin
      -- some code using Swap
   end Sort;

end Sorting;
```

Implementations of nested procedures

Several known implementations:

- In functional or object-oriented languages:
  - full-fledged heap-allocated closures
  - more general than nested procedures
- In languages that obey a stack discipline, classical techniques are rather tricky:
  - "static links" (in the P-code machine [Wirth 1966])
  - "displays" [Dijkstra 1961]

Two optimized implementations but no high-level semantics!

A verified implementation of nested procedures

- We formalized a frame stack as an Abstract Data Type

A verified implementation of nested procedures

- We formalized a frame stack as an Abstract Data Type

Definition I := *nat* $\times$ *nat*.

Structure *FS* $\{$*V* : *Type*$\}$:=
$\{$
S : *Type*;
empty: S;
fetch: $S \to I \to$ *option V*;
update: $S \to I \to V \to$ *option* S;
top_frame: $S \to$ *option* (*list* V);
new_frame: $S \to$ *nat* $\to$ *list* $V \to$ (*option* S);
clear_frame: $S \to S \to$ *nat* $\to$ *option* S;
frame_offset: $S \to I \to$ *option nat*
$\}$.

A verified implementation of nested procedures

- We formalized a frame stack as an Abstract Data Type

A verified implementation of nested procedures

- We formalized a frame stack as an Abstract Data Type
- We provided two implementations of this ADT:
  - a simple high-level implementation (our prototype)
  - an optimized implementation based on “static links”

A verified implementation of nested procedures

- We formalized a frame stack as an Abstract Data Type
- We provided two implementations of this ADT:
  - a simple high-level implementation (our prototype)
  - an optimized implementation based on "static links"
- We proved in Coq that the optimized implementation is correct with respect to the prototype, by defining a bi-simulation.

A verified implementation of nested procedures

- We formalized a frame stack as an Abstract Data Type
- We provided two implementations of this ADT:
  - a simple high-level implementation (our prototype)
  - an optimized implementation based on "static links"
- We proved in Coq that the optimized implementation is correct with respect to the prototype, by defining a bi-simulation.
- This bi-simulation then gives us for free a strong property called "parametricity" [Reynolds 1983].
- Parametricity in implemented in Coq as a plugin [Keller & Lasson 2012]

A verified implementation of nested procedures

- As a corollary of parametricity, you obtain the following informal property:

  for any programming language,
  for any semantics relying on the frame stack ADT,
  the optimized implementation works as expected

A verified implementation of nested procedures

- As a corollary of parametricity, you obtain the following informal property:

  for any programming language,
  for any semantics relying on the frame stack ADT,
  the optimized implementation works as expected

- You need to provide the syntax and the semantics of your language, and Coq does the rest:
  You get a formal machine-checked proof of this property.

A verified implementation of nested procedures

- As a corollary of parametricity, you obtain the following informal property:

  for any programming language,
  for any semantics relying on the frame stack ADT,
  the optimized implementation works as expected

- You need to provide the syntax and the semantics of your language, and Coq does the rest:
  You get a formal machine-checked proof of this property.

- Some statistics (just for nested procedures)
  - 1,000 lines of statement
  - 2,000 lines of proof

Future Works

- Full SPARK 2014 support (packages, generics, ...)
- Correctness of SPARK tools (static analysis, contracts, ...)
- Correctness of the OCaml compiler (and its runtime)?
- Correctness of the Coq proof assistant?
- ...

What is now the weakest link in the chain?