# Pôle Preuves et langages

Jean-Christophe Filliâtre

19 octobre 2022

LMF
Laboratoire
Méthodes
Formelles

1. structure
2. scientific activities
3. focus (15 minutes) : A. Paskevich and J.-H. Jourdan
   Proof of programs with Why3 and Creusot
4. focus (15 minutes) : K. Nguyễn
   Occurence Typing and Set-theoretic types
   for dynamic languages

# members

- 26 permanent members


Idir

Thibaut

Bruno

Véronique

Frédéric

Valentin

Sylvie

Frédéric

Hubert

Sylvain

Évelyne

Gilles

Jean-Christophe

Armaël

Jacques-Henri

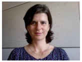
Chantal

Claude

Guillaume

Kim

Andrei

Christine

Mihaela

Safouan

Théo

Burkhart

Lina

- 4 postdocs and engineers
- 24 PhD students

- 4 teams
  - Arithmétique des ordinateurs (G. Melquiond)
  - Calcul, langages et compilation (K. Nguyễn)
  - Preuve de programmes (A. Paskevich)
  - Preuve mécanisée (C. Keller)
- 2 Inria EPC
  - Deducteam (G. Dowek)
  - Toccata (C. Marché)

Arithmétique des ordinateurs

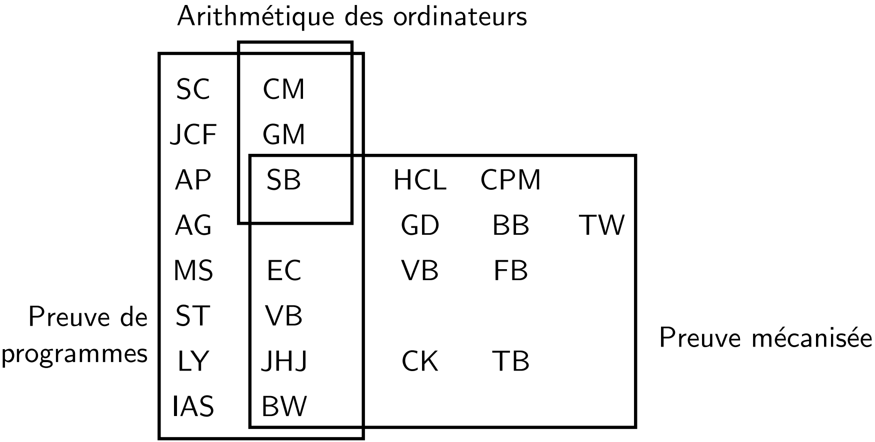
CM
GM
SB

# structure

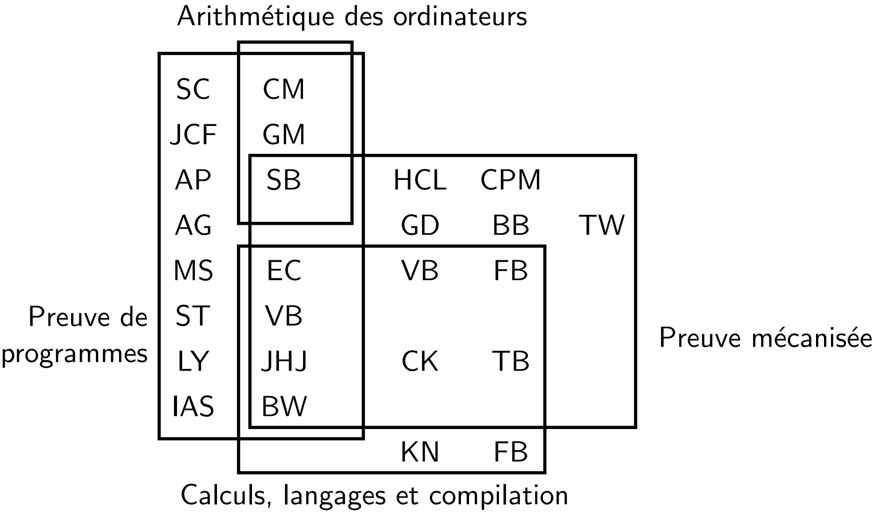
Arithmétique des ordinateurs

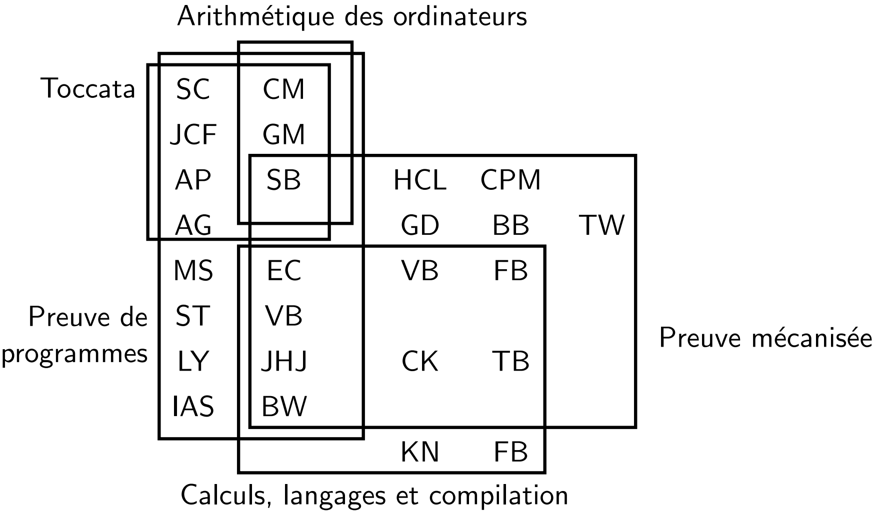
| | | |
|---|---|---|
| | SC | CM |
| | JCF | GM |
| | AP | SB |
| | AG | |
| | MS | EC |
| Preuve de programmes | ST | VB |
| | LY | JHJ |
| | IAS | BW |

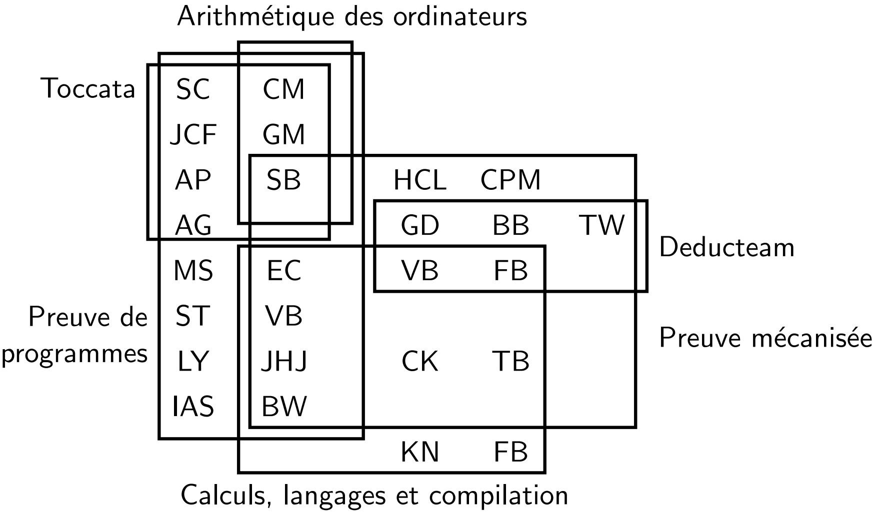
# structure


Arithmétique des ordinateurs
SC
JCF
AP
AG
MS
ST
LY
IAS
CM
GM
SB
EC
VB
JHJ
BW
HCL
CPM
GD
BB
TW
VB
FB
CK
TB
Preuve de programmes
Preuve mécanisée

# structure


Arithmétique des ordinateurs
SC
JCF
AP
AG
MS
ST
LY
IAS
CM
GM
SB
EC
VB
JHJ
BW
HCL
CPM
GD
BB
TW
VB
FB
CK
TB
KN
FB
Preuve de programmes
Preuve mécanisée
Calculs, langages et compilation


Arithmétique des ordinateurs
Toccata
SC
CM
JCF
GM
AP
SB
HCL
CPM
AG
GD
BB
TW
MS
EC
VB
FB
Preuve de programmes
ST
VB
Preuve mécanisée
LY
JHJ
CK
TB
IAS
BW
KN
FB
Calculs, langages et compilation


Arithmétique des ordinateurs
Toccata
SC
CM
JCF
GM
AP
SB
HCL
CPM
AG
GD
BB
TW
Deducteam
MS
EC
VB
FB
Preuve de programmes
ST
VB
Preuve mécanisée
LY
JHJ
CK
TB
IAS
BW
KN
FB
Calculs, langages et compilation

Responsable : Guillaume Melquiond

- IEEE 754 floating-point numbers
- GMP-like big integers
- interval arithmetic

✍ Sylvie Boldo, Guillaume Melquiond.
*Some Formal Tools for Computer Arithmetic : Flocq and Gappa.*
ARITH 2021

Sylvie Boldo co-leader GDR IM's GT ARITH

Responsable : Kim Nguyễn

- formal semantics
- study and design of languages
- programming and domain-specific languages

✍ Valentin Blot.
*A direct computational interpretation of second-order arithmetic via update recursion*. LICS 2022.

Responsable : Andrei Paskevich

- deductive program verification
- reasoning on mutable memory
- verification of properties of systems

✍ Cláudio Belo Lourenço Denis Cousineau, Florian Faissole, Claude Marché, David Mentré, Hiroaki Inoue.
*Automated Verification of Temporal Properties of Ladder Programs*. Formal Methods for Industrial Critical Systems 2021.

Responsable : Chantal Keller

- proof systems
- interaction between proof systems
- formalized libraries

✍ Frédéric Blanqui, Gilles Dowek, Émilie Grienenberger, Gabriel Hondet, François Thiré.
*Some Axioms for Mathematics*. FSCD 2021

# Ongoing grants

## International

- EMC2 [ERC Synergy →2025]
- EuroProofNet [COST →2025]
- FRESCO [ERC Consolidator →2026]

## National

- France–Japan [Sakura →2023]
- Gospel [ANR →2026]
- ICSPA [ANR →2024]
- NARCO [ANR →2026]
- NuSCAP [ANR →2025]
- SecurEval
  [PEPR Cybersécurité →2027]

## Industrial

- AdaCore [bilatéral →2023]
- Langages dynamiques pour les données
  [Oracle Labs →2022]
- Mitsubishi Electric R&D
  [bilatéral →2023]
- TrustInSoft [bilatéral →2023]
- thèses CIFRE
  [OCamlPro, Tarides →2023]

Mainly at LMF :

- Alt-Ergo
- CoqInterval
- Coq-num-analysis
- Coquelicot
- CDuce
- Creusot
- Cubicle
- Datacert
- Datalogcert
- Dedukti
- Ekstrakto
- Flocq
- Gappa
- Lambdapi
- OntoEventB
- SMTCoq
- Sniper
- Why3
- WhyMP

LMF also contributes to :

- Coq
- HOL-TestGen
- Iris
- Isabelle
- OCaml

# PhD students

In preparation :

- Léo Andrès [+OCamlPro]
- Luc Chabassier
- Xavier Denis
- Louise Dubois De Prisque
- Thiago Felicissimo
- Valentin Fouillard [+LISN]
- Paul Geneau de Lamarlière [+MERCE]
- Yoan Géran
- Emilie Grienenberger
- Baptiste Gueuziec [+CEA]
- Alexandrina Korneva
- Antoine Lanco
- Mickael Laurent [+U Paris Cité]
- Amélie Ledein
- Nicolas Meric
- Glen Mevel [+Inria]
- Josué Moreau
- Houda Mouhcine [+LIPN +Inria Paris]
- Clément Pascutto [+Tarides]
- Quentin Petitjean [+LIG]
- Andrew Samokhish [+K. Inside]
- Julien Simonnet [+CEA]
- Rébecca Zucchini

Defended :

- Yacine El Haddad [09/21]
- Gaspard Ferey [06/21]
- Diane Gallois-Wong [+LIP6] [03/21]
- Quentin Garchery [01/22]
- Guillaume Girol [+CEA] [10/22]
- Gabriel Hondet [09/22]
- Yaëlle Vinçont [12/21, +CEA]

# Proof of programs with WHY3 and CREUSOT

Andrei Paskevich    Jacques-Henri Jourdan

LMF — October 19, 2022

# Deductive program verification

```
(*  Given an array a of signed integers, compute the maximum sum s *)
(*  of its contiguous subarray. The sum of an empty subarray is 0. *)


val maximum_subarray (a: array int): (s: int)
```

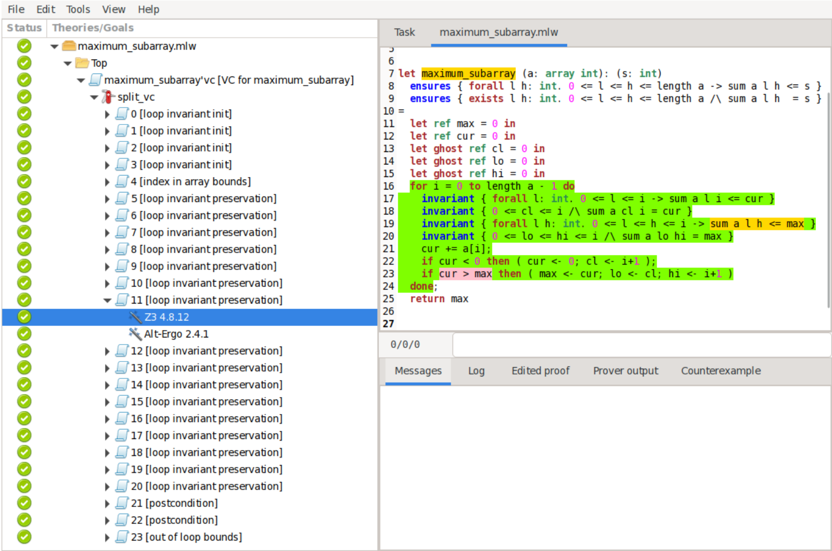
```
(*  Given an array a of signed integers, compute the maximum sum s *)
(*  of its contiguous subarray. The sum of an empty subarray is 0. *)



let maximum_subarray (a: array int): (s: int)
=
  let ref max = 0 in
  let ref cur = 0 in
  for i = 0 to length a - 1 do
    cur += a[i];
    if cur < 0 then cur <- 0;
    if cur > max then max <- cur;
  done;
  return max
```

```
(*  Given an array a of signed integers, compute the maximum sum s *)
(*  of its contiguous subarray. The sum of an empty subarray is 0. *)

let maximum_subarray (a: array int): (s: int)
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= s }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = s }
=
  let ref max = 0 in
  let ref cur = 0 in
  for i = 0 to length a - 1 do
    cur += a[i];
    if cur < 0 then cur <- 0;
    if cur > max then max <- cur;
  done;
  return max
```

```
(*  Given an array a of signed integers, compute the maximum sum s *)
(*  of its contiguous subarray. The sum of an empty subarray is 0. *)

let maximum_subarray (a: array int): (s: int)
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= s }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = s }
=
  let ref max = 0 in
  let ref cur = 0 in




  for i = 0 to length a - 1 do






    cur += a[i];
    if cur < 0 then ( cur <- 0 );
    if cur > max then ( max <- cur )
  done;
  return max
```

```
(*  Given an array a of signed integers, compute the maximum sum s *)
(*  of its contiguous subarray. The sum of an empty subarray is 0. *)

let maximum_subarray (a: array int): (s: int)
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= s }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = s }
=
  let ref max = 0 in
  let ref cur = 0 in
  let ghost ref cl = 0 in
  let ghost ref lo = 0 in
  let ghost ref hi = 0 in
  for i = 0 to length a - 1 do
    invariant { forall l: int. 0 <= l <= i -> sum a l i <= cur }
    invariant { 0 <= cl <= i /\ sum a cl i = cur }
    invariant { forall l h: int. 0 <= l <= h <= i -> sum a l h <= max }
    invariant { 0 <= lo <= hi <= i /\ sum a lo hi = max }
    cur += a[i];
    if cur < 0 then ( cur <- 0; cl <- i+1 );
    if cur > max then ( max <- cur; lo <- cl; hi <- i+1 )
  done;
  return max
```

# WHY3 proof session



# WHY3 in a nutshell

WHYML, a programming language

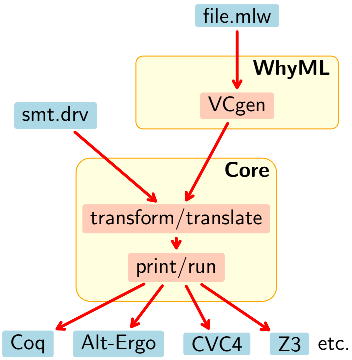
- type polymorphism • variants
- limited support for higher order
- pattern matching • exceptions
- integer & FP computer arithmetic
- mutable data with controlled aliasing
- ghost code and ghost data

WHY3, a program verification tool

- VC generation using WP or SP
- 70+ VC *transformations* ($\approx$ tactics)
- support for 25+ ATP and ITP systems
- produces OCaml and C code

WHYML, a specification language

- polymorphic & algebraic types
- limited support for higher order
- inductive predicates



Three different ways of using WHY3

- as a logical language
  - a convenient front-end to many theorem provers
  - probabilistic programs: EASYCRYPT (Barthe et al.)
- as a programming language for correct-by-construction software
  - see examples in our gallery
    `https://toccata.gitlabpages.inria.fr/toccata/gallery/why3.en.html`
  - Multiple Precistion arithmetic library: WHYMP (Melquiond Rieu-Helft)
- as an intermediate verification language
  - Ada programs: SPARK (AdaCore)
  - C programs: FRAMA-C (Marché Moy)
  - Java programs: KRAKATOA (Marché Paulin Urbain)
  - Rust programs: CREUSOT (Denis Jourdan Marché)

# WhyMP: arbitrary-precision integer arithmetic

The GNU Multiple Precision arithmetic library (GMP)

- free software, widely used
- state-of-the-art algorithms, unmatched performances

# WHYMP: arbitrary-precision integer arithmetic

The GNU Multiple Precision arithmetic library (GMP)

- free software, widely used
- state-of-the-art algorithms, unmatched performances
- highly intricate algorithms written in low-level C and ASM
- ill-suited for random testing

  GMP 5.0.4: *"Two bugs in multiplication [. . . ] with extremely low probability [. . . ]. Two bugs in the gcd code [. . . ] For uniformly distributed random operands, the likelihood is infinitesimally small."*

# WhyMP: arbitrary-precision integer arithmetic

## The GNU Multiple Precision arithmetic library (GMP)

- free software, widely used
- state-of-the-art algorithms, unmatched performances
- highly intricate algorithms written in low-level C and ASM
- ill-suited for random testing

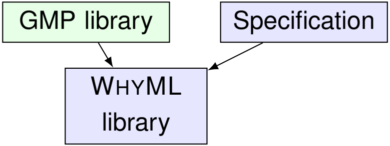
  GMP 5.0.4: *"Two bugs in multiplication [. . . ] with extremely low probability [. . . ]. Two bugs in the gcd code [. . . ] For uniformly distributed random operands, the likelihood is infinitesimally small."*

## Objectives

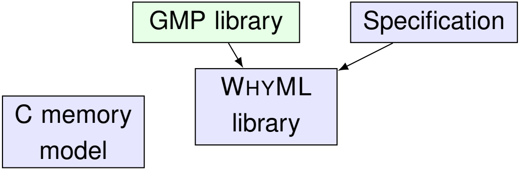
- produce a verified library compatible with GMP
- attain performances comparable to a no-assembly GMP

GMP library

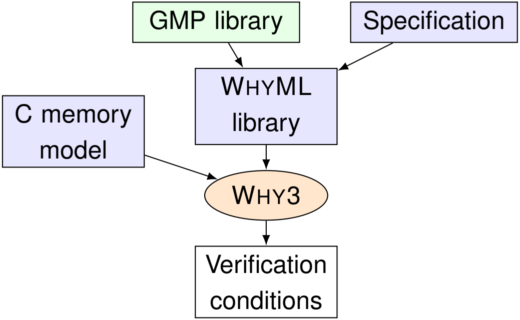
# The WHY3 workflow



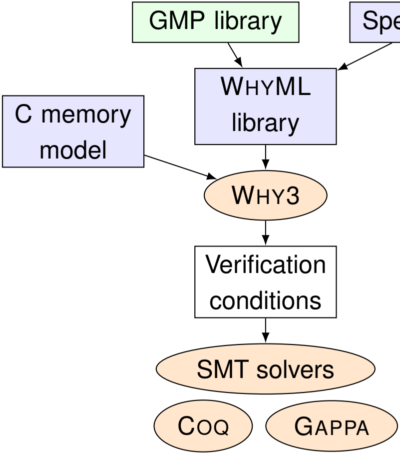
# The WHY3 workflow



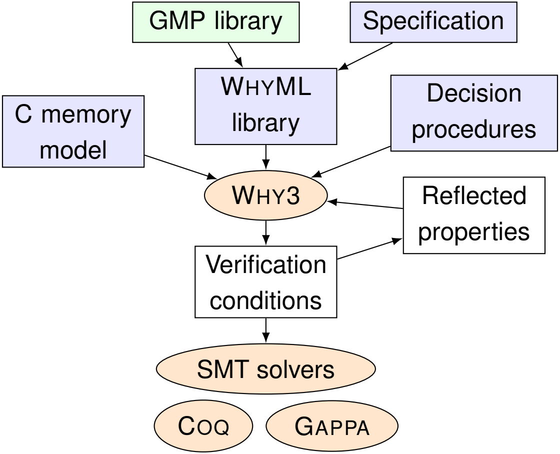
# The WHY3 workflow



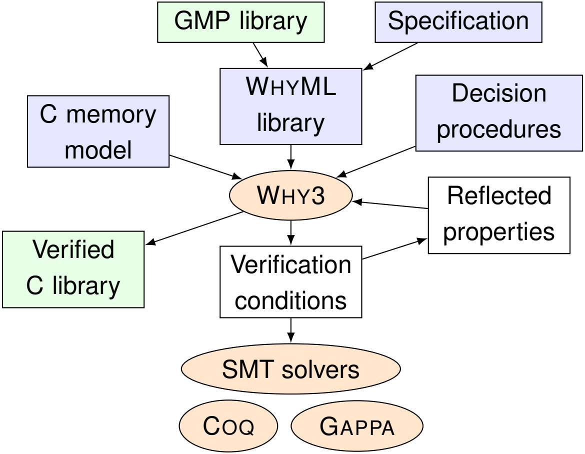
# The WHY3 workflow



# The WHY3 workflow



# The WHY3 workflow



# Example: subtracting 1 from a long integer

Original macro (simplified from 18-line mpn_decr_u)

```
#define mpn_decr_1(p)          \
  mp_ptr __p = (p);            \
  while ((*(__p++))-- == 0) ;
```

# Example: subtracting 1 from a long integer

## Original macro (simplified from 18-line mpn_decr_u)

```
#define mpn_decr_1(p)          \
  mp_ptr __p = (p);            \
  while ((*(__p++))-- == 0) ;
```

## Conversion to WhyML

```
let wmpn_decr_1 (p: ptr uint64) : unit
=
  let ref lp = 0 in
  while lp = 0 do
    lp <- get p;
    set p (sub_mod lp 1);
    p <- p ++ 1;
  done
```

# Example: subtracting 1 from a long integer

## Original macro (simplified from 18-line `mpn_decr_u`)

```
#define mpn_decr_1(p)          \
  mp_ptr __p = (p);            \
  while ((*(__p++))-- == 0) ;
```

## Conversion to WhyML

```
let wmpn_decr_1 (p: ptr uint64) (ghost sz: int32) : unit
  requires { valid p sz }
  requires { 1 <= value p sz } (* <- real bug in GMP *)
  ensures  { value p sz = value (old p) sz - 1 }
=
  let ref lp = 0 in
  while lp = 0 do
    lp <- get p;
    set p (sub_mod lp 1);
    p <- p ++ 1;
  done
```

# Example: subtracting 1 from a long integer

## Original macro (simplified from 18-line mpn_decr_u)

```
#define mpn_decr_1(p)          \
  mp_ptr __p = (p);            \
  while ((*(__p++))-- == 0) ;
```

## Extraction to C

```
void wmpn_decr_1(uint64_t * p) {
  uint64_t lp = 0;
  while (lp == 0) {
    lp = *p;
    *p = lp - 1;
    p = p + 1;
  }
}
```

# A Verified Library

22 000 lines of WhyML code — 5 000 lines of extracted C code

Performance: comparable to GMP without assembly code

Supported operations:

- addition, subtraction, comparison
- multiplication: quadratic, Toom-Cook 2, Toom-Cook 2.5
- division: “schoolbook”
- square root: divide-and-conquer
- modular exponentiation with Montgomery reduction
- base conversion and input-output

# A new systems programming language: Rust

High performance, high level of control with safe abstraction mechanisms and concurrency features.

Tends to replace C/C++ for many sensitive applications:

- Web browsers (Firefox),
- Cloud infrastructures (Amazon),
- Operating systems kernels (Linux),
- Embedded systems

How to guarantee correctness of Rust code?

# Our claim

Rust is well-suited for verification.

It has a rich type system.

- Safety guarantees.
  - No need to prove safety.
- Non-aliasing guarantees.
  - Helps reasoning with pointers.
- Type traits (i.e., type classes): abstraction and genericity mechanism.
  - Easy to write reusable specifications.

Creusot, a deductive verification tool for Rust.

- The user annotates Rust code with specifications.
- Creusot translates Rust into WhyML (Why3 = Creusot's back-end).
- Exploits type system (non-aliasing, genericity with traits, safety).

One crucial aspect of Creusot: lightweight handling of pointers.

Our big secret:

**Rust is a purely functional* programming language**

... and this greatly helps verification.

*some squinting required.

```rust
fn incr(mut x: u64, y: u64) -> (u64, u64) {
  x += y;
  do_stuff();
  (x, y)
}
```

```ocaml
let incr x y =
  let x = x + y in
  do_stuff ();
  (x, y)
```

As long as there is no pointer, mutating variables = shadowing.

We give a name to the value of each variable at each program point.
Variables can only be mutated directly, we can track their modifications.

`Box<T>` is Rust's type for owned pointers.

- I.e., pointers that are allocated on the heap ($\simeq$ `malloc`).
- Type system guarantee: no alias.

```
fn incr(x: Box<u64>, y: Box<u64>)
      -> (Box<u64>, Box<u64>) {
  *x += *y;
  do_stuff();
  (x, y)
}
```

```
let incr x y =
  let x = x + y in
  do_stuff();
  (x, y)
```

Because there is no alias, we know nobody else can mutate `x` and `y`.

Owned pointers functionally behave like their content.

Rust has another notion of pointer: borrows.

- Temporary pointer to a memory location.
- They are crucial to effective use of Rust.

Their handling in Creusot requires a new mechanism: prophecies.

- Prophecies allow concise specifications.
- But their soundness cause subtle causality issues.
  - We have a strong meta-theoretical formalization.

# Occurence Typing and Set-theoretic types for dynamic languages

Kim NGUYỄN

19 octobre 2022

# Motivations

## A function in JavaScript

```
function foo(x) {
  return (typeof(x) === "number")? x+1 : x.trim();
  }
```

Idiomatic in dynamic languages:

- JavaScript: optional arguments (`typeof(y) === "undefined"`)
- JavaScript: poor person's overloading
- Python: 5400 dynamic type tests in the standard library
- Python: `isinstance(x,(int, float, str))`

# Motivations

## A function in TypeScript (Microsoft)/Flow (Facebook)

```
function foo(x : number | string) : number | string {
  return (typeof(x) === "number")? x+1 : x.trim();
  }
```

# Motivations

## A function in TypeScript (Microsoft)/Flow (Facebook)

```
function foo(x : number | string) : number | string {
  return (typeof(x) === "number")? x+1 : x.trim();
  }
```

$(\texttt{number|string}) \rightarrow (\texttt{number|string})$

# Motivations

## A function in TypeScript (Microsoft)/Flow (Facebook)

```
function foo(x : number | string) : number | string {
  return (typeof(x) === "number")? x+1 : x.trim();
  }
```

$$(\texttt{number|string}) \to (\texttt{number|string})$$

We present a type system with occurrence typing and:

# Motivations

## A function in TypeScript (Microsoft)/Flow (Facebook)

```
function foo(x : number | string) : number | string {
  return (typeof(x) === "number")? x+1 : x.trim();
  }
```

(number|string) → (number|string)

We present a type system with occurrence typing and:

- Expressive types: (number → number) ∧ (string → string) (e.g., `foo(42)+1`)

# Motivations

## A function in TypeScript (Microsoft)/Flow (Facebook)

```
function foo(x : number | string) : number | string {
  return (typeof(x) === "number")? x+1 : x.trim();
  }
```

$(\texttt{number|string}) \to (\texttt{number|string})$

We present a type system with occurrence typing and:

- Expressive types: $(\text{number} \to \text{number}) \wedge (\text{string} \to \text{string})$ (e.g., `foo(42)+1`)
- Inference (no need for user-defined type annotations)

# Motivations

## A function in JavaScript

```
function foo(x) {
  return (typeof(x) === "number")? x+1 : x.trim();
  }
```

$(\texttt{number|string}) \to (\texttt{number|string})$

We present a type system with occurrence typing and:

- Expressive types: $(\mathsf{number} \to \mathsf{number}) \wedge (\mathsf{string} \to \mathsf{string})$ (e.g., `foo(42)+1`)
- Inference (no need for user-defined type annotations)

# Motivations

## A function in JavaScript

```
function foo(x) {
  return (is_number_or_function(x))? x+1 : x.trim();
  }
```

$(\texttt{number|string}) \to (\texttt{number|string})$

We present a type system with occurrence typing and:

- Expressive types: $(\text{number} \to \text{number}) \wedge (\text{string} \to \text{string})$ (e.g., `foo(42)+1`)
- Inference (no need for user-defined type annotations)
- Works for checks with arbitrary expressions (*eg*, applications of user-defined functions)

# Language

## Functional language

$$
\begin{array}{rrcl}
\textbf{Types} & t & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid t \to t
\end{array}
$$

$$
\begin{array}{rrcl}
\textbf{Expressions} & e & ::= & c \mid x \mid \lambda x.e \mid ee \\
\textbf{Values} & v & ::= & c \mid \lambda x.e
\end{array}
$$

Call-by-value semantics:

$$(\lambda x.e)v \quad \rightsquigarrow \quad e\{v/x\}$$

# Language

## Functional language with set-theoretic types

$$\textbf{Types} \quad t \quad ::= \quad \mathrm{Int} \mid \mathrm{Bool} \mid t \to t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathtt{Any}$$

$$\begin{array}{rccl} \textbf{Expressions} & e & ::= & c \mid x \mid \lambda x.e \mid ee \\ \textbf{Values} & v & ::= & c \mid \lambda x.e \end{array}$$

Call-by-value semantics:

$$(\lambda x.e)v \quad \rightsquigarrow \quad e\{v/x\}$$

# Language

Functional language with set-theoretic types and typecases

$$\textbf{Types} \quad t \quad ::= \quad \mathsf{Int} \mid \mathsf{Bool} \mid t \to t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathtt{Any}$$

$$\begin{array}{rcll} \textbf{Expressions} & e & ::= & c \mid x \mid \lambda x.e \mid ee \mid (e{\in}t)\,?\,e : e \\ \textbf{Values} & v & ::= & c \mid \lambda x.e \end{array}$$

Call-by-value semantics:

$$(\lambda x.e)v \quad \rightsquigarrow \quad e\{v/x\}$$

# Language

## Functional language with set-theoretic types and typecases

$$\textbf{Types} \quad t \quad ::= \quad \mathsf{Int} \mid \mathsf{Bool} \mid t \to t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathtt{Any}$$

$$\begin{array}{rccl} \textbf{Expressions} & e & ::= & c \mid x \mid \lambda x.e \mid ee \mid (e{\in}t)\,?\,e : e \\ \textbf{Values} & v & ::= & c \mid \lambda x.e \end{array}$$

Call-by-value semantics:

$$\begin{array}{rcll} (\lambda x.e)v & \leadsto & e\{v/x\} & \\ (v{\in}t)\,?\,e_1 : e_2 & \leadsto & e_1 & \text{if } v \in t \\ (v{\in}t)\,?\,e_1 : e_2 & \leadsto & e_2 & \text{if } v \notin t \end{array}$$

# Type System

## Occurrence Typing and Typecases Part

$$[\in_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e{\in}t)\ ?\ e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e{\in}t)\ ?\ e_1 : e_2 : t_2}$$

$$[\vee]\ \frac{\Gamma \vdash e' : t_1 \vee t_2 \qquad \Gamma, x : t_1 \vdash e : t \qquad \Gamma, x : t_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

# Type System

## Occurrence Typing and Typecases Part

$$[\in_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e{\in}t)\ ?\ e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e{\in}t)\ ?\ e_1 : e_2 : t_2}$$

$$[\vee]\ \frac{\Gamma \vdash e' : t_1 \vee t_2 \qquad \Gamma, x : t_1 \vdash e : t \qquad \Gamma, x : t_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

### Rationale

- $[\vee]$ splits the type of the checked expression as necessary
- $[\in_i]$ distribute the appropriate split to each branch

# Type System

## Occurrence Typing and Typecases Part

$$[\in_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e{\in}t)\ ?\ e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e{\in}t)\ ?\ e_1 : e_2 : t_2}$$

$$[\vee]\ \frac{\Gamma \vdash e' : t_1 \vee t_2 \qquad \Gamma, x : t_1 \vdash e : t \qquad \Gamma, x : t_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

### Rationale (i.e., the essence of occurrence typing)

- $[\vee]$ splits the type of the checked expression as necessary
- $[\in_i]$ distribute the appropriate split to each branch

# Type System

## Occurrence Typing and Typecases Part

$$[\in_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e{\in}t)\ ?\ e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e{\in}t)\ ?\ e_1 : e_2 : t_2}$$

$$[\vee]\ \frac{\Gamma \vdash e' : t_1 \vee t_2 \qquad \Gamma, x : t_1 \vdash e : t \qquad \Gamma, x : t_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

**Rationale (i.e., the essence of occurrence typing)**

- $[\vee]$ splits the type of the checked expression as necessary
- $[\in_i]$ distribute the appropriate split to each branch

Example: For $f : (\mathrm{Int} \to (\mathrm{Int} \vee \mathrm{Bool})) \wedge (\neg\mathrm{Int} \to \mathrm{Bool})$

$\lambda x.(f\ x{\in}\mathrm{Int})\ ?\ (f\ x) + x + 1 : 42$ has type $(\mathrm{Int}{\to}\mathrm{Int}) \wedge (\neg\mathrm{Int}{\to}42)$

# Towards an algorithmic system

## What is not algorithmic in $[\vee]$?

$$[\vee]\ \frac{\Gamma \vdash e' : t_1 \vee t_2 \qquad \Gamma, x : t_1 \vdash e : t \qquad \Gamma, x : t_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

- It can be applied anywhere (on any expression)
- It can use any subexpression
- It can substitute any occurrence(s) of this subexpression

# Towards an algorithmic system

## What is not algorithmic in [$\vee$]?

$$[\vee]\ \frac{\Gamma \vdash e' : t_1 \vee t_2 \qquad \Gamma, x : t_1 \vdash e : t \qquad \Gamma, x : t_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

- It can be applied anywhere (on any expression)
- It can use any subexpression
- It can substitute any occurrence(s) of this subexpression

- It can use any decomposition of the type of $e'$

# Towards an algorithmic system

## What is not algorithmic in $[\vee]$?

$$[\vee]\ \frac{\Gamma \vdash e' : t_1 \vee t_2 \qquad \Gamma, x : t_1 \vdash e : t \qquad \Gamma, x : t_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

- It can be applied anywhere (on any expression)
- It can use any subexpression
- It can substitute any occurrence(s) of this subexpression

Not syntax directed
$\Rightarrow$ *Normal forms*

- It can use any decomposition of the type of $e'$

Not analytic
$\Rightarrow$ Annotations

# Maximal-Sharing Canonical Form

[$\simeq$ A-normal form on steroids, used for typing only]

- Apply [$\vee$] on **every** subexpression
  $\Rightarrow$ Define each subexpression in a separate definition
- Substitute **all** occurrences of a subexpression
  $\Rightarrow$ Two occurrences of the same expression share the **same definition**

# Maximal-Sharing Canonical Form

[$\simeq$ A-normal form on steroids, used for typing only]

- Apply [$\vee$] on **every** subexpression
  $\Rightarrow$ Define each subexpression in a separate definition
- Substitute **all** occurrences of a subexpression
  $\Rightarrow$ Two occurrences of the same expression share the **same definition**

$$
\begin{array}{l}
MSCF(\lambda x.(f\ x{\in}\mathsf{Int})\,?\,((f\ x)+x+1):42) = \\
\qquad \lambda x.\begin{array}[t]{l}
\texttt{bind}\ u = x+1\ \texttt{in} \\
\texttt{bind}\ v = f\,x\ \texttt{in} \\
\texttt{bind}\ w = v+u\ \texttt{in} \\
\texttt{bind}\ y = (v{\in}\mathsf{Int})\,?\,w:42\ \texttt{in} \\
y
\end{array}
\end{array}
$$

# Maximal-Sharing Canonical Form

[$\simeq$ A-normal form on steroids, used for typing only]

- Apply [$\vee$] on **every** subexpression
  $\Rightarrow$ Define each subexpression in a separate definition
- Substitute **all** occurrences of a subexpression
  $\Rightarrow$ Two occurrences of the same expression share the **same definition**

| | | | |
|---|---|---|---|
| **Atomic expressions** | $\boldsymbol{a}$ | $::=$ | $c \mid \lambda x.\boldsymbol{\kappa} \mid (x, x) \mid xx \mid (x{\in}\tau)\,?\,x : x \mid$ |
| **Canonical Forms** | $\boldsymbol{\kappa}$ | $::=$ | $x \mid \texttt{bind}\, x = \boldsymbol{a}\ \texttt{in}\ \boldsymbol{\kappa}$ |

[$\vee$] now becomes:

$$[\vee]\ \frac{\Gamma \vdash e' : t_1 \vee t_2 \qquad \Gamma, x : t_1 \vdash e : t \qquad \Gamma, x : t_2 \vdash e : t}{\Gamma \vdash \texttt{bind}\, x = e'\ \texttt{in}\ e\ : t}$$

# Annotations

## Example

$$
\begin{array}{l}
MSCF(\lambda x.(f\ x{\in}\mathsf{Int})\ ?\ ((f\ x) + x + 1) : 42) = \\
\qquad \lambda x : \{\mathsf{Int}, \neg\mathsf{Int}\}\ . \\
\qquad\quad \mathtt{bind}\ u : \{(x{:}\mathsf{Int}){\triangleright}\mathsf{Int}\} = x + 1\ \mathtt{in} \\
\qquad\quad \mathtt{bind}\ v : \{\mathsf{Int}, \mathsf{Bool}\} = f\ x\ \mathtt{in} \\
\qquad\quad \mathtt{bind}\ w : \{(u{:}\mathsf{Int}, v{:}\mathsf{Int}){\triangleright}\mathsf{Int}\} = v + u\ \mathtt{in} \\
\qquad\quad \mathtt{bind}\ y : \{\mathsf{Int}, 42\} = (v{\in}\mathsf{Int})\ ?\ w : 42\ \mathtt{in} \\
\qquad\quad y
\end{array}
$$

# Annotations

## Example

$$\begin{array}{l} MSCF(\lambda x.(f\ x{\in}\mathsf{Int})\ ?\ ((f\ x)+x+1):42) = \\ \quad \lambda x : \{\mathsf{Int}, \neg\mathsf{Int}\}\,. \\ \qquad \mathtt{bind}\ u : \{(x{:}\mathsf{Int}){\triangleright}\mathsf{Int}\} = x+1\ \mathtt{in} \\ \qquad \mathtt{bind}\ v : \{\mathsf{Int}, \mathsf{Bool}\} = f\ x\ \mathtt{in} \\ \qquad \mathtt{bind}\ w : \{(u{:}\mathsf{Int}, v{:}\mathsf{Int}){\triangleright}\mathsf{Int}\} = v+u\ \mathtt{in} \\ \qquad \mathtt{bind}\ y : \{\mathsf{Int}, 42\} = (v{\in}\mathsf{Int})\ ?\ w:42\ \mathtt{in} \\ \qquad y \end{array}$$

*Results:*

- $e$ typable $\iff$ $MSCF(e)$ typable $\iff$ annotated-$MSCF(e)$ typable
- Transforming into MSCF is effective
- Type-checking for annotated MSCFs is decidable

# Annotations

## Example

$$MSCF(\lambda x.(f\,x{\in}\mathrm{Int})\,?\,((f\,x)+x+1):42) =$$

$$\begin{array}{l}\lambda x : \{\mathrm{Int}, \neg\mathrm{Int}\}\,.\\ \quad \texttt{bind}\ u : \{(x{:}\mathrm{Int}){\triangleright}\mathrm{Int}\} = x+1\ \texttt{in}\\ \quad \texttt{bind}\ v : \{\mathrm{Int}, \mathrm{Bool}\} = f\,x\ \texttt{in}\\ \quad \texttt{bind}\ w : \{(u{:}\mathrm{Int}, v{:}\mathrm{Int}){\triangleright}\mathrm{Int}\} = v+u\ \texttt{in}\\ \quad \texttt{bind}\ y : \{\mathrm{Int}, 42\} = (v{\in}\mathrm{Int})\,?\,w:42\ \texttt{in}\\ \quad y\end{array}$$

*Results:*

- $e$ typable $\iff$ $MSCF(e)$ typable $\iff$ annotated-$MSCF(e)$ typable
- Transforming into MSCF is effective
- Type-checking for annotated MSCFs is decidable

**Is $e$ typable?**

If possible to annotate $MSCF(e)$ to give it some type $t$, then $\vdash e : t$.
Otherwise, $e$ is not typable.

# Annotations

## Example

$$
\begin{array}{l}
MSCF(\lambda x.(f\,x{\in}\mathsf{Int})\,?\,((f\,x)+x+1):42) = \\
\qquad \lambda x : \{\mathsf{Int}, \neg\mathsf{Int}\}\,. \\
\qquad\quad \texttt{bind}\ u : \{(x{:}\mathsf{Int}){\triangleright}\mathsf{Int}\} = x+1\ \texttt{in} \\
\qquad\quad \texttt{bind}\ v : \{\mathsf{Int}, \mathsf{Bool}\} = f\,x\ \texttt{in} \\
\qquad\quad \texttt{bind}\ w : \{(u{:}\mathsf{Int}, v{:}\mathsf{Int}){\triangleright}\mathsf{Int}\} = v+u\ \texttt{in} \\
\qquad\quad \texttt{bind}\ y : \{\mathsf{Int}, 42\} = (v{\in}\mathsf{Int})\,?\,w:42\ \texttt{in} \\
\qquad\quad y
\end{array}
$$

*Results:*

- $e$ typable $\iff$ $MSCF(e)$ typable $\iff$ annotated-$MSCF(e)$ typable
- Transforming into MSCF is effective
- Type-checking for annotated MSCFs is decidable

### Is $e$ typable?

**We defined and implemented a sound algorithm that determines annotations using an analysis of the applications and typecases**

# Examples from our prototype

```
type Falsy = False | "" | 0
type Truthy = ~Falsy
```

# Examples from our prototype

```
type Falsy = False | "" | 0
type Truthy = ~Falsy
```

```
let not_ x =
  if x is Truthy then false
                 else true
```

```
let to_Bool x = not_ (not_ x)
```

```
let and_ x y =
  if x is Truthy then to_Bool y
                 else false
```

```
let or_ x y =
  not_ (and_ (not_ x) (not_ y))
```

# Examples from our prototype

```
type Falsy = False | "" | 0
type Truthy = ~Falsy
```

```
let not_ x =
  if x is Truthy then false
                 else true
```

$$\left.\right\}\begin{array}{l}(\mathsf{Truthy} \to \mathsf{False}) \wedge \\ (\mathsf{Falsy} \to \mathsf{True})\end{array}$$

```
let to_Bool x = not_ (not_ x)
```

$$\left.\right\}\begin{array}{l}(\mathsf{Truthy} \to \mathsf{True}) \wedge \\ (\mathsf{Falsy} \to \mathsf{False})\end{array}$$

```
let and_ x y =
  if x is Truthy then to_Bool y
                 else false
```

$$\left.\right\}\begin{array}{l}(\mathsf{Falsy}\to\mathsf{Any}\to\mathsf{False})\wedge \\ (\mathsf{Truthy}\to\mathsf{Truthy}\to\mathsf{True})\wedge \\ (\mathsf{Truthy}\to\mathsf{Falsy}\to\mathsf{False})\end{array}$$

```
let or_ x y =
  not_ (and_ (not_ x) (not_ y))
```

$$\left.\right\}\begin{array}{l}(\mathsf{Truthy}\to\mathsf{Any}\to\mathsf{True})\wedge \\ (\mathsf{Falsy}\to\mathsf{Truthy}\to\mathsf{True})\wedge \\ (\mathsf{Falsy} \to \mathsf{Falsy} \to \mathsf{False})\end{array}$$

# Current and future work

This work (collaboration with Giuseppe Castagna (IRIF) and Mickaël Laurent (IRIF & LMF)) has been presented at POPL22. Since then

- Polymorphism:

```
concat : [ 'a * ] -> [ 'b* ] -> [ 'a* 'b* ]
filter : (('a & 'b) -> True & ('a \ 'b) -> False) -> ['a*] ->
     [('a&'b)*]
```

- Started work on integrating side effects (with Thibaut Balabonski and Philippe Volte–Viera, PEPR Secureval).