On the translation of Maude programs to modern imperative languages

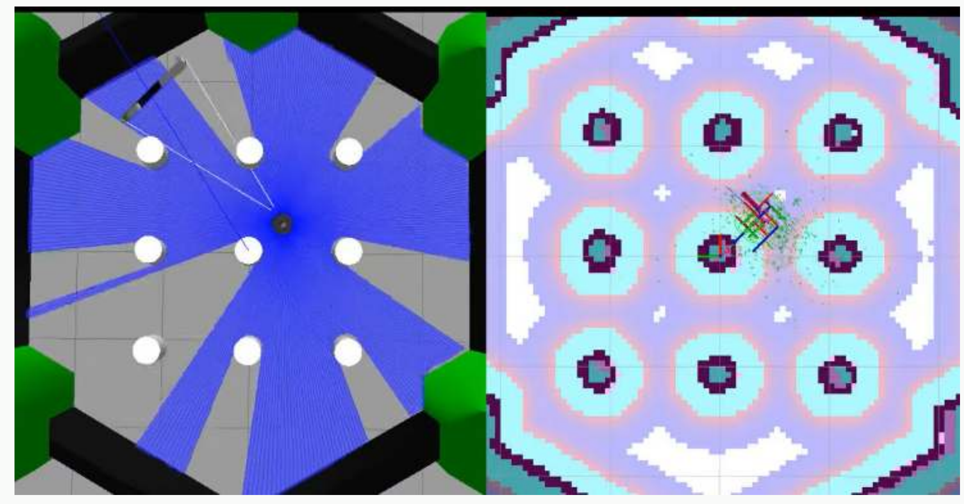
Rubén Rubio · Beatriz Alcaide García · Adrián Riesco

Universidad Complutense de Madrid

WRLA 2026 · April 11, Torino, Italia

Motivation

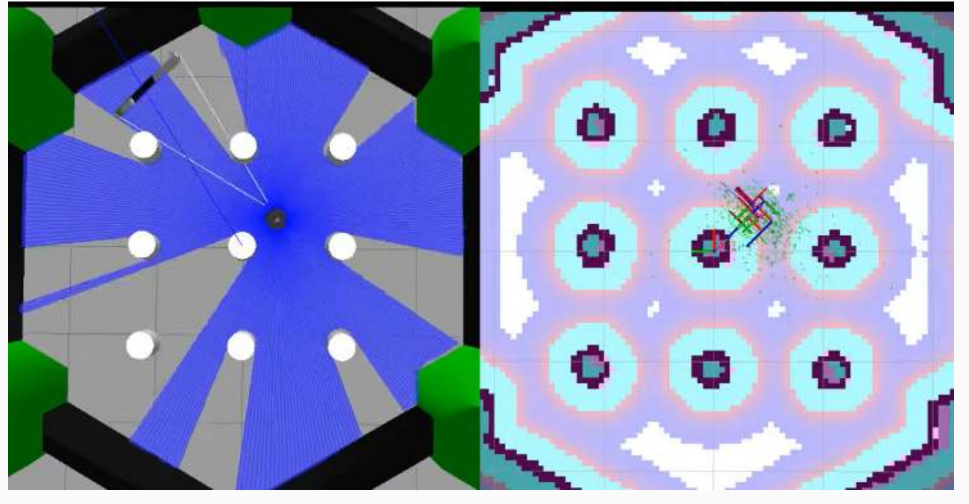
Integration



External objects · (Python) bindings

Motivation

Integration



External objects · (Python) bindings

Performance

Statistical model checking

many executions + numeric

Related work

- Abstract machines (1994-2001): Reduce ELAN Machine, Rewrite Rule Machine, Term Rewriting Abstract Machine (CafeOBJ)
- Compilation of Maude to SIMD and MIMD machines
- Semicompilation in Maude (pattern automata, discrimination nets)

Related work

- Abstract machines (1994-2001): Reduce ELAN Machine, Rewrite Rule Machine, Term Rewriting Abstract Machine (CafeOBJ)
- Compilation of Maude to SIMD and MIMD machines
- Semicompilation in Maude (pattern automata, discrimination nets)

Translation to C — ELAN, ASF+SDF (Rascal), Soufflé

Related work

- Abstract machines (1994-2001): Reduce ELAN Machine, Rewrite Rule Machine, Term Rewriting Abstract Machine (CafeOBJ)
- Compilation of Maude to SIMD and MIMD machines
- Semicompilation in Maude (pattern automata, discrimination nets)

Translation to C — ELAN, ASF+SDF (Rascal), Soufflé

Translation between formalisms — Hets, Maude2Lean, MTT, etc.

From Maude …

Tutorial

LNCS 4350

Manuel Clavel Francisco Durán
Steven Eker Patrick Lincoln
Narciso Martí-Oliet José Meseguer
Carolyn Talcott

All About Maude –
A High-Performance
Logical Framework

How to Specify, Program and Verify
Systems in Rewriting Logic

📕 Garavel, Tabikh, Arrada. *Benchmarking implementations of term rewriting and pattern matching…*. WRLA 2018

From Maude …

Manuel Clavel Francisco Durán
Steven Eker Patrick Lincoln
Narciso Martí-Oliet José Meseguer
Carolyn Talcott

Tutorial

LNCS 4350

All About Maude – A High-Performance Logical Framework

How to Specify, Program and Verify Systems in Rewriting Logic

📕 Garavel, Tabikh, Arrada. *Benchmarking implementations of term rewriting and pattern matching…*. WRLA 2018

Highly expressive

- User-defined syntax
- Sorts and memberships
- Structural axioms (A, C, AC, …)
- Conditionals involving searchs
- Non-deterministic rewriting
- Hash-consing

… to modern imperative languages

“Algebraic datatypes”

Pattern matching

… to modern imperative languages

“Algebraic datatypes”

Pattern matching

Algebraic data types

Haskell

```haskell
data Maybe a = Nothing | Just a
```

Rust

```rust
enum Option<T> { None, Some(T) }
```

Recursive algebraic data types

Haskell

```haskell
data Stack = Empty | Push Int Stack
```

Rust

```rust
enum Stack { Empty, Push(i64, Stack) }
```

Recursive algebraic data types

Haskell

```
data Stack = Empty | Push Int Stack
```

Rust

```
enum Stack { Empty, Push(i64, Box<Stack>) }
```

explicit memory management

Pattern matching constructs

Haskell

```haskell
case value of
  Just(x) -> Just(2 * x)
  Nothing -> Nothing
```

Rust

```rust
match value {
  Some(x) => Some(2 * x),
  None    => None,
}
```

Pattern matching constructs — Common features

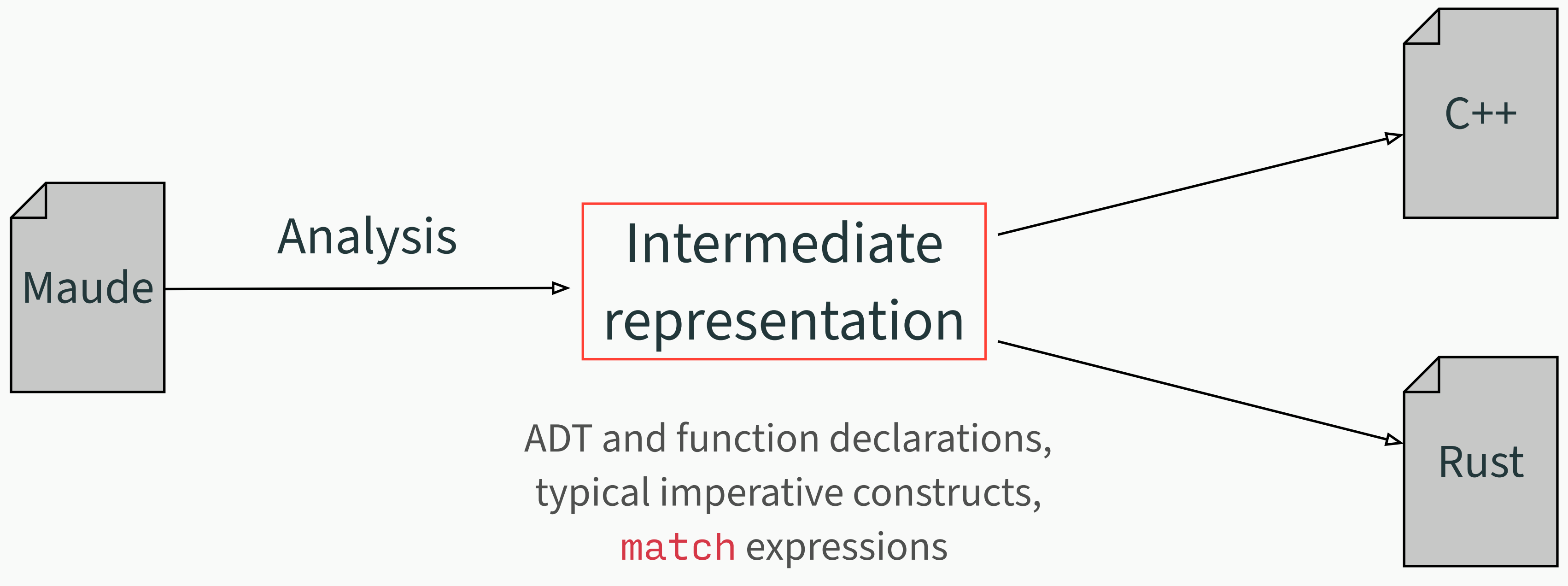
```
match expression {
  ...
  case pattern if guard => body
  ...
}
```

- Nesting
- Linearity (no repeated variables)
- Guards (conditions for each case)
- Disjunctions of patterns ($p_1 \mid \cdots \mid p_n$)
- Matching of predefined data structures (lists, tuples, etc.)

Principles of the translation

1. Exploit target language features
   - Pattern matching and ADTs
   - Predefined data structures
2. Multiple languages through an intermediate representation
3. Derivation rather than dull code generation
4. Pragmatical goals

Translation



Translation of functional modules

```
fmod module is

  sorts sorts .
  subsorts subsorts .

  mb membership axioms .

  op constructor [ctor] .
  op defined operator .

  eq equation .

endfm
```

- fmod module is → program in the target language
- kind → algebraic datatype
- mb membership axioms → sort predicate
- op constructor [ctor] → case in ADT definition → algebraic datatype

Translation of functional modules

```
fmod module is

  sorts sorts .
  subsorts subsorts .

  mb membership axioms .

  op constructor [ctor] .
  op defined operator .

  eq equation .

endfm
```

- `fmod module is` → program in the target language
- kind → algebraic datatype
- `mb membership axioms .` → sort predicate
- `op constructor [ctor] .` → case in ADT definition → algebraic datatype
- `op defined operator .` → function with a match
- `eq equation .` → case in its function → function with a match

A simple stack (example)

```
fmod STACK is
  protecting INT .

  sorts Stack NeStack .
  subsort NeStack < Stack .

  op empty : -> Stack [ctor] .
  op push : Int Stack
              -> NeStack [ctor] .

  eq length : Stack -> Nat .

  var N : Int . var S : Stack .

  eq length(empty) = 0 .
  eq length(push(N, S)) =
    s length(S) .
endfm
```

Rust

```
enum Stack {
  Empty,
  Push(i64, Box<Stack>),
}

fn is_NeStack(stack: &Stack) -> bool
{
  matches!(stack,
     Stack::Push(_, _))
}
```

A simple stack (example)

```
fmod STACK is
  protecting INT .

  sorts Stack NeStack .
  subsort NeStack < Stack .

  op empty : -> Stack [ctor] .
  op push : Int Stack
            -> NeStack [ctor] .

  eq length : Stack -> Nat .

  var N : Int . var S : Stack .

  eq length(empty) = 0 .
  eq length(push(N, S)) =
    s length(S) .
endfm
```

Rust

```rust
fn length(stack: &Stack) -> i64
{
  match stack {
    Stack::Empty => 0,
    Stack::Push(_, s) =>
      length(*s) + 1,
  }
}
```

A simple stack (example)

```
fmod STACK is
  protecting INT .

  sorts Stack NeStack .
  subsort NeStack < Stack .

  op empty : -> Stack [ctor] .
  op push : Int Stack
            -> NeStack [ctor] .

  eq length : Stack -> Nat .

  var N : Int . var S : Stack .

  eq length(empty) = 0 .
  eq length(push(N, S)) =
    s length(S) .
endfm
```

C++

```
struct Stack :
  public std::variant<
    std::monospace,
    std::tuple<int, smart_ptr<Stack>>
  >
{
  enum Variant { EMPTY, PUSH };
};
```

A simple stack (example)

```
fmod STACK is
  protecting INT .

  sorts Stack NeStack .
  subsort NeStack < Stack .

  op empty : -> Stack [ctor] .
  op push : Int Stack
              -> NeStack [ctor] .

  eq length : Stack -> Nat .

  var N : Int . var S : Stack .

  eq length(empty) = 0 .
  eq length(push(N, S)) =
    s length(S) .
endfm
```

C++

```
int length(const Stack& stack)
{
  if (stack.index() == Stack::EMPTY)
    return 0;

  if (stack.index() == Stack::PUSH)
    return 1 + length(*std::get<1>(
      std::get<Stack::PUSH>(stack)
    ));
}
```

Translation of structural axioms

Relies on

- Special constructor in the ADT of its kind
- Normalizing constructor
- Pattern matching specialization

Iterated	`f : S -> S [iter]`	`F(S, i64)`
Commutative	`f : S S -> T [comm]`	$f(p_1, p_2) \mid f(p_2, p_1)$
Associativity	`f : S S -> S [assoc]`	Lists
AC(U)	`f : S S -> S [assoc comm]`	Multisets

Translation of structural axioms (C) (example)

```
op minimum : Nat Nat -> Nat [comm] .
vars M N : Nat .
ceq minimum (M, N) = M if M <= N .
```

```
fn minimum(n: i64 , m: i64) -> i64 {
  match (n, m) {
    (n, m) | (m, n) if m <= n => m,
  }
}
```

Translation of structural axioms (A) (example)

```
fn palindrome(s: &[Symbol]) -> bool {
  match s {
    [] => true ,  // palindrome(nil) = true
    [_] => true , // palindrome(S) = true

    // palindrome(S L S) = palindrome(L)
    [s1 , l @ .., s2] if s1 == s2 => palindrome(l),

    // palindrome(S1 L S2) = false
    [_, .., _] => false ,
  }
}
```

Translation of rules (on top)

Share much with equation translation.

```
fn rlapp_kind(term: &kind) -> Option<kind> {
  match term {
    ...
    case lhs if condition => Some(rhs)
    ...
    _ => None,
  }
}
```

```
crl lhs => rhs if condition .
```

Example of rule translation

```
mod REWRITE is
  sorts Foo Bar .

  op f : Bar Foo -> Foo [ctor] .
  ops a b c : -> Bar [ctor] .

  rl [ab] : a => b .
  rl [bc] : b => c .
endm
```

```
fn rlapp_Bar(term: &Bar)
    -> Option<Bar>
{
  match term {
    Bar::A => Some(Bar::B),
    Bar::B => Some(Bar::C),
    _ => None,
  }
}
```

Translation of rules (everywhere)

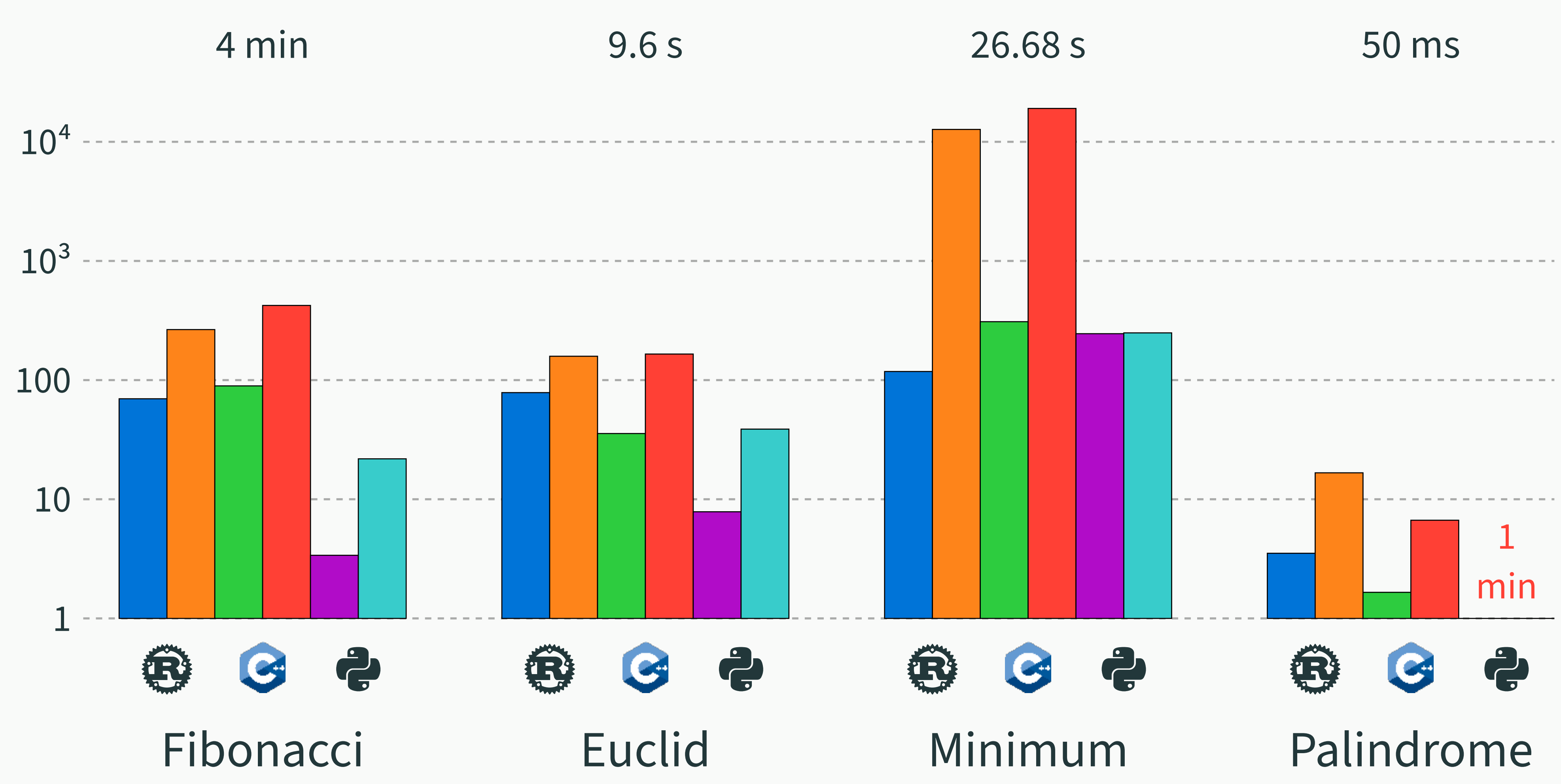
```
fn arlapp_kind(term: &kind) -> Option<kind> {
  match term {
    ...
    kind::F(.., x, ..) => {
      ...
      if let Some(rx) = arlapp_kindx(x)
            { return F(.., rx, ..) }
      ...
    }
    ...
  }
  rlapp_kind(term)
}
```

Limitations

Not yet supported:

- AC and general A patterns `[elem, .. @ list, elem]`
- Non-deterministic rewriting
- Search in rewriting conditions
- Strategies
- Specialization for object-oriented rewriting

Benchmarks

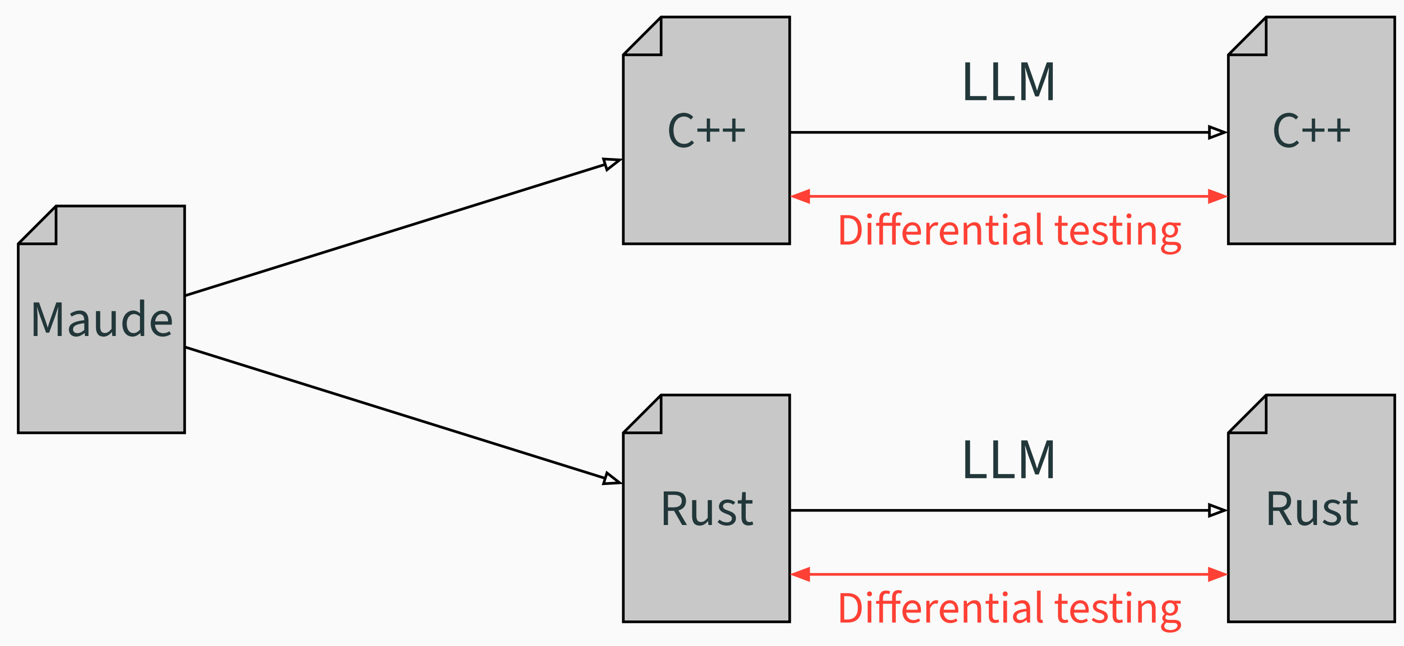


Conclusions and future work

- Compiler from (a subset of) Maude to imperative programming language
- Exploiting ADT and pattern matching support
- Likely performance improvement, but needs more experiments

- Implement the missing features
- Optimization and more derivation-like features

Improvement of code with LLM



📕 R. Rubio, B. Alcaide-García, R. Riesco.
Improving generated programs using large language models. PROLE 2026

Thank you

github.com/fadoss/maudec