

# Chandeliers

A Lustre-in-Rust compiler

Neven Villani

2024-01-10

`github.com:vanille-n/chandeliers`

# An introduction to Rust and Proc Macros

# About Rust

- **compiled** language
- strong **type system**
- extensible via **macros**

## Extending Rust with macros

- Custom parser,
- Arbitrary code execution,
- Unsanitized identifiers.

What I did for Lustre is an instance of a more general fact: you can embed inside Rust **any language** if it “agrees” with Rust on

- types, ownership and safety → language must be **memory-safe**
- tokens and parentheses → macro expansion is **post-tokenization**

# The Rust ecosystem

“crate” ~ library/package → published on <https://crates.io>

rustc: official compiler

cargo: package manager

# What is a macro ?

Different invocations:

- `#[derive(...)]`
- `println!(...)`

Different declarations:

- `macro_rules!`
- `#[proc_macro]`

Common characteristic: mapping *`Fn(TokenStream)`*  $\rightarrow$  `TokenStream`

## A standard macro

```
use std::collections::HashMap;
```

```
#[derive(Default)]
```

```
struct Thing {
```

```
    n: usize,
```

```
    map: HashMap<char, f64>,
```

```
    label: Option<String>,
```

```
}
```

```
fn main() {}
```

## Expanded

```
$ cargo expand
```

```
impl ::core::default::Default for Thing {  
    fn default() → Thing {  
        Thing {  
            n: ::core::default::Default::default(),  
            map: ::core::default::Default::default(),  
            label: ::core::default::Default::default(),  
        }  
    }  
}
```



## In short

- **Macros are regular functions** `TokenStream → TokenStream`
- Procedural Macros can execute **arbitrary code** at compile-time (“proc macros”)

→ Chandeliers consists of **one macro** that contains a parser, typechecker, and code generator

# Chandeliers quick guide

# Structure of a program using Chandeliers

```
# Cargo.toml
[dependencies]
chandeliers-lus = "0.5"

// main.rs
use chandeliers_lus::decl;

// Rust glue code
decl! {
    // Lustre code → expanded to equivalent Rust code
}
```

## Example

```
// main.rs
chandeliers_lus::decl! {
  node counting() returns (n : int);
  let
    n = 0 fby n + 1;
  tel;
}
// [...]
```

Every node is expanded to (at least) one *struct* with a step function.

# Annotations

Rust-style attributes `#[ ... ]`

Some of the most useful:

- `#[trace("foo({x}) = {y}")]`  
node foo(x : int) returns (y : int);
- `#[main(100)]`  
node main() returns ();
- `#[export]` and `#[pub]` levels of visibility
- `#[doc("Add node documentation here")]`

# Advantages

## What we get (almost) for free

- good **performance** through LLVM
- strong **typing guarantees** (hard to make mistakes in glue code)
- good **error messages** and dead code analysis
- **IDE** integration (rust-analyzer, clippy, ...)
- glue code can **import crates**
- Lustre libraries are Rust libraries
  - can be uploaded to `crates.io` and downloaded by `cargo`
  - documentation available on `docs.rs`
  - builtin test framework available (nodes annotated `#[test]`)

# A typical error message

```
node foo(m : int) returns (f : float);
```

```
let f = m; tel;
```

error: Type mismatch between the left and right sides:  
Base types should be unifiable: expected float, got int

→ src/lib.rs:605:13

```

|
605 |         let f = m; tel;
|           ^^^^^
|

```

note: This element has type float

→ src/lib.rs:605:13

```

|
605 |         let f = m; tel;
|           ^
|

```

note: While this element has type int

→ src/lib.rs:605:17

```

|
605 |         let f = m; tel;
|           ^
|

```



# Error in glue code

```
chandeliers_lus::decl! {
  #[export]
  node foo() returns (n: int);
  let n = 0; tel;
}
```

```
chandeliers_lus::decl! {
  extern node foo() returns (n: float);
}
```

```
error[E0308]: mismatched types
  ──> src/lib.rs:609:21
   |
609 |         extern node foo() returns (n: float);
   |                                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |                                |               |
   |                                |               expected `Nillable<f64>` because of return type
   |                                expected `Nillable<f64>`, found `Nillable<i64>`
   |
   = note: expected enum `Nillable<f64>`
           found enum `Nillable<i64>`
```

# Using external crates

```
use rand::{rngs::ThreadRng, Rng};

use chandeliers_sem::traits::{Embed, Step};
use chandeliers_sem::{implicit_clock, ty};

/// Lustre node that returns a random `int` uniformly between
/// `i64::MIN` and `i64::MAX`.
#[derive(Debug, Default, Clone)]
pub struct random_int {
    /// Internal random number generator.
    rng: ThreadRng,
}

impl Step for random_int {
    type Input = ();
    type Output = i64;
    fn step(&mut self, __inputs: ty!()) → ty!(int) {
        implicit_clock!(__inputs);
        self.rng.gen::<<i64>>().embed()
    }
}
```

**Difficulties encountered**

## Parser limitations

- no control over the tokenizer
  - program must be well-parenthesized (not an issue)
  - comments must be Rust-style: `// ...` and `/* ... */`
  - Rust reserved keywords can't be used as Lustre variables
- syn handles left recursion poorly
  - manual management of associativity and precedence
  - bugs in the parser lead to rustc stack overflow

## API constraints

- macro output must be self-contained
- 1 node = 1 step function (glue code requires stable API)
- no `null` in Rust → Chandeliers works with *Option*

## Lack of test frameworks

- most frameworks are optimized for unit tests, not proc macros
- `trybuild`, the most complete, doesn't support `fail` tests  
(only `compile-fail`)

## The bad side of error messages

- Rust's error messages are optimized for the end user, not for the macro developers.
- `panic` in macro execution prints a very vague error

```
#[warn(clippy::expect_used,  
      clippy::panic,  
      clippy::unreachable,  
      clippy::unwrap_used)]
```

- getting the Spans to show the right location is sometimes quite brittle

# A full example

(coding demo)



# General porting procedure

0. `cargo new`
1. Create `Cargo.toml` and depend on `chandeliers-std, sem, lus`
2. wrap code in `chandeliers_lus::decl! { ... }`
3. rename variables if they conflict with Rust reserved keywords
4. fix Chandeliers-specific semantic choices
5. add annotations
  - `#[main]`, `#[test]` on your toplevel functions
  - `#[trace(...)]` everywhere relevant